

MSX-C入門 上巻

桜田幸嗣 著

アスキー出版局

商 標

MS-DOS は米国マイクロソフト社の登録商標です。

MSX, MSX2, MSX2+, MSX-DOS, MSX-DOS2, MSX-C は株式会社アスキーの商標です。

UNIX オペレーティング・システムは、AT&T のベル研究所が開発し、AT&T がライセンスしています。

はじめに

コンピュータというものは、つくづくイケナイ機械だと思います。なぜって、コンピュータで遊んでいると、どうしてあんなにも時を忘れて熱中してしまうのでしょうか。なにもゲームにハマっているわけではないのです。もちろん、それ(良質のゲーム)で、風薫る初夏の休日を見事に潰すようなことだってあります。はい、否定はいたしません。

しかし、ここで言いたいのは、プログラミングという作業の危険なまでの面白さなのです。寝る前に、ほんのちょいと、睡眠薬がわりにプログラムを1本モノにしよう、などと考えたのが運の尽き、気が付けば夜は白々と明け始め、それに比例して目は真っ赤、ああ今日も徹夜してしまった……。しかし、そのむなしさの中でも、大脳左半球はプログラミングの興奮にうち震えている。この喜びを知るあなた、そんなあなたには、ぜひ本書を読んでもらいたい。そして共にC言語を語りあかしたいものです。

よく耳にするC言語の売り文句は、いわく構造化されたコンパイラであり、したがってプログラムは作りやすい上に実行スピードも速い、また効率的なマシンコードを生成できるためシステムプログラミングにも最適で、現にあのUNIXはCで書かれているんだぞお、などというところでしょうか(言ってる意味がわからなくても気にせぬように)。たしかにウソではありません。

しかし、我々MSXユーザーが、なぜMSX-Cを使うのかと問われたなら、その理由はこういった大層なものでは決してないでしょう。MSXのユーザーは、だいたいにおいて、コンピュータを遊びの対象と考えています。プログラミングとは仕事のために行うものではなく、あくまで自分の好奇心を満足させるために行うものと考えています。みなさんもそうでしょうか？

そんな我々にとって、MSX-Cは、なによりもまずプログラミングというものの楽しさを教えてくれる道具であります。なぜMSX-Cを使うのか、その答えは簡単、それがとんでもなく面白いからの一言です。ソフトウェアサイエンスの世界には、多くの巧妙な考え方やアルゴリズムがキラ星のように

はじめに

散在しています。こんなにもエレガントな方法があったのかっ！ そう認識した瞬間は、何者にも換えがたい喜びがあります。その境地への架け橋となるのが、ほかならぬ C 言語なのです。BASIC？ そんなもの、C 言語の魅力の前にはメじゃないですぜ(とはいえやっぱり BASIC も好きなのですからね、私は)。

本書では上下巻の2冊を通して、この MSX-C によるプログラミングの面白さをみなさんにお伝えできるように努力したつもりです。さあ、C 言語を覚えましょう。そして、さらなるプログラミングの世界に身をゆだねましょう。徹夜明けの、あの黄金の太陽が、みなさんの頭上に輝かんことを！

平成元年 六月吉日 著者記す

■ 上下巻の構成

この「MSX-C 入門」は、上下巻の2分冊で構成されています。各巻の内容は以下に示すとおりです。

●上 巻

MSX-C Ver.1.1/Ver.1.2 を対象とした C 言語の入門編です。ただしページ数の都合もあり、ここで C 言語のすべてが語られているわけではありません。この上巻では、ほんとうに必要な最低限と思われる C の文法を中心に内容をまとめてみました。実際の C プログラミングを行う上でのテクニックについては、下巻にはいつから実践的に身に付けていただきます。

●下 巻

MSX のグラフィック機能やサウンド機能を日一杯に使いこなすための実践編です。スクリーンモード1の画面を使ったバックマン風の迷路追いかけゲームと、スクリーンモード5の画面を使ったグラフィックもぐら叩きゲームの2本のプログラムをメインの柱に据えて、MSX-C を使う上でのさまざまな手法を解説していきます。なお前者は旧タイプの MSX でも実行できますが、後者には MSX2 以降の機種が必要です。

目 次

はじめに	3
------------	---

1章

MSX-C ノ ススメ 11

1.1 どうして MSX-C を使うのか	13
----------------------------	----

2章

プログラミングの環境を整える 21

2.1 これだけそろえば準備 OK	23
2.2 インストールの手順	26
■ DOS1 のためのインストール手順	26
■ DOS2+Ver.1.1 のためのインストール手順	28
■ DOS2+Ver.1.2 のためのインストール手順	32
■ 最後のチェック(全バージョン)	36

3章

コンパイラの動作とその使い方 37

3.1 ソースプログラムの準備	39
■ MSX-C の 4 段活用	40
■ CF の動作	41
■ CG の動作	43
■ M80 の動作	45
■ L80 の動作	47
3.2 CC コマンドによるコンパイル	50
■ 「CC.BAT」の作成	50
■ 再び「WC.C」のコンパイル	52

4章

これがCのプログラムだ

53

4.1 Cのプログラムに触れてみる	55
■ プログラムの入力から実行まで	55
■ Cのプログラムの合体パワー	63
4.2 Cのプログラムの読み方	67
■ プログラムの外見を拝見	67
■ 関数がプログラムを組み立てる	70
■ 用意周到の宣言文	75
■ コメントは「/ *」と「*/」で囲む	80

5章

基本的なプログラミング

83

5.1 画面にデータを出力する	85
■ 1つの文字を表示する — putchar()	85
■ 文字列を表示する — puts()	86
■ 数値を表示する — printf()	88
■ 改行を行う — ニューライン文字	89
5.2 変数とデータの計算法	91
■ 文字変数の登場 — char 型の変数	91
■ 数値を扱う変数 — int 型の変数	92
■ 配列変数の使い方 — Cの配列表現	93
■ 加減乗除を行う — Cの算術演算記号	95
■ ビット操作を行う — Cのビット演算記号	98
■ 文字データに手を加える — char 型変数の操作	100
5.3 キーボードから入力する	103
■ 押されたキーをすぐに読む — getch()	103
■ 入力と出力のカラミ合い — バッファリングの問題	106
■ 文字列を入力する — gets()	110

6章

Cのプログラムは如何にして組み立てられるのか 113

6.1 条件判断	115
■ プログラムの最小要素 一文	115
■ 条件の判定 if 文	116
■ 文をまとめる 複文	119
■ データの大小を比較する 比較演算子	121
6.2 複雑な条件判断	124
■ 条件判断のしくみ 論理値	124
■ 条件を組み合わせる 論理演算子	125
■ 代入した値を判定する 代入式	128
6.3 繰り返し	129
■ 回数を指定した繰り返し for 文	129
■ 空ループによる時間かせぎ 空文	132
■ 省エネ方式の代入文 代入演算子	133
■ 条件が成立するまで繰り返す while 文と do 文	134
6.4 それ以外の制御構造	138
■ ループから抜け出す break 文	138
■ ループの処理を1回だけスキップする continue 文	140
■ ループの奥底からの脱出 goto 文	141
■ 効果的な場合分け処理 switch 文	143

7章

数値データの扱い方 147

7.1 数値のいろいろな書き表し方	149
■ 4種類の数値表現	149
■ 16進数や8進数の表示	151
7.2 数値データの型と、その利用法	154
■ データの型とは何か	154
■ 3種類のデータの型	156
■ データ型の整合性チェック	162

8章

画面表示を工夫する

167

- 8.1 フォーマット指定付きの数値表示169
 - 数値の桁ぞろえ表示169
 - フォーマット指定付きの数値表示172
- 8.2 コントロール文字による画面制御175
 - 特別な記号を持つコントロール文字175
 - 文字コードで表すコントロール文字176
- 8.3 エスケープシーケンス178
 - エスケープシーケンスを使ってみる178
 - カーソル位置の指定179
 - 行の挿入と削除180
 - カーソルのちらつき防止182
- 8.4 そのほかの話題183
 - グラフィック文字の表示183
 - クォート記号や¥記号の表示184

9章

関数は新しい世界を開くか

187

- 9.1 関数の作成法とその使い方189
 - 関数を定義する189
 - 戻り値を返す関数202
- 9.2 記憶クラスとスコープ205
 - 変数の記憶クラス205
 - 変数のスコープ207

10章

ポインタと配列と文字列 211

10.1 ポインタを使ったプログラム	213
■ アドレス演算子「&」と間接演算子「*」	213
■ 配列とポインタの関係	218
■ 関数定義にポインタを利用する	221
10.2 文字列の操作	225
■ 文字列の正体を調べる	225
■ 文字列操作の実例	227
10.3 構造化データを使う	229
■ 構造体の使い方	229

11章

データの保存と利用 233

11.1 ファイルを扱うプログラム	235
■ データ読み書きの基本型	235
■ 読み書きの手段のバリエーション	242
■ 特別なファイルの利用法	244

Appendix 1 MSX-C エラーメッセージ	249
---------------------------------	-----

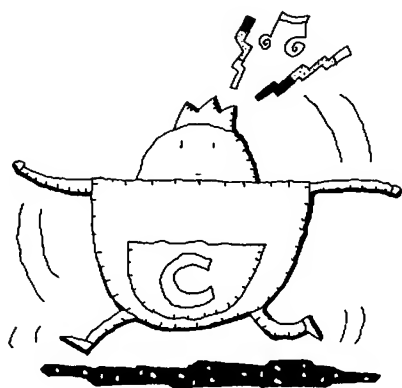
Appendix 2 VRAM ディスクの使い方	253
--------------------------------	-----

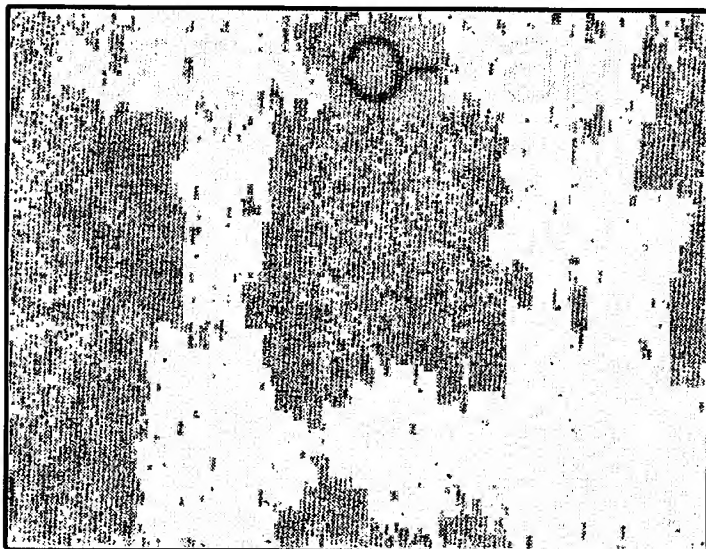
索引	259
----------	-----

参考文献および推薦書籍	263
-------------------	-----

1章

MSX-C ノススメ







どうして MSX-C を 使うのか

この章では MSX-C の持つメリットを他のプログラミング言語と比較しながら簡単にまとめてみました。MSX-C は本当に役に立つのだろうか、そんな不安と期待をお持ちのみなさん、これを読んで安心して、おおらかな気持ちで MSX-C の勉強をはじめてください。

そんなことどうでもいいから早く MSX-C の使い方を教えてっ、というせっかちな方はこの章をとばして 2 章へどうぞ。

●手軽だが遅い BASIC

MSX のユーザーの大多数は、たぶん BASIC によってプログラミングの世界(あるいは泥沼)への第一歩を踏み出すのだと思います。読者の皆さんもそうだったでしょう。筆者も例外ではありません。

この BASIC というやつは、じつに便利なプログラミング言語です。プログラムは簡単に作れるうえに、計算はもちろん絵も描ければ音楽も演奏できる。MSX をいじるなら、BASIC さえあれば、ほかになんにもいらないと思ったりします。でも、それは初めのうちだけで、ある程度 BASIC を使い込めるとだんだん限界も目につき不満がつのってきます。その筆頭が「BASIC は遅い」ということでしょう。データベースでも、ゲームでも、BASIC で自作したプログラムは市販の同様なものにくらべると反応がワンテンポどころかスリーテンポぐらい遅れ、どうにも使い心地がよくありません。

●マシン語は高速だが面倒

そんなとき、まず思い付くのはマシン語の利用です。なにしろマシン語で書かれたプログラムの実行スピードはすこぶる速い。BASIC の数倍から十数倍にも達します。

しかし、その代わりマシン語の扱いは BASIC よりはるかに面倒でもあり

ます。なぜって、マシン語というのは非常にコマカイのですね。たとえば BASIC なら画面に線を引くのは LINE 命令一発で OK ですが、同じことをマシン語でプログラムしようとすると、やれ点の位置を計算してデータをどこそこに格納してと、つまらない手間ばかりかかります。おまけにマシン語は針の先ほどのミスがあっても暴走してしまうため、プログラミング中いつも注意力を凝らしておかなくてはなりません。マシン語はスピードアップの特効薬には違いありませんが、この薬は副作用が多すぎるのが玉にキズなのです。

●コンパイラは手軽で速い

そこで登場するのが、MSX-C をはじめとするコンパイラです。コンパイラはマシン語以外のプログラムをオートマチックにマシン語に書き直してくれます。つまり高速なプログラムが労せずして手にはいるわけです。

参考までに BASIC と MSX-C のスピード比較結果を 1 つ紹介しましょう。ここでは四則演算や条件判断が適当に含まれた簡単な例ということで、友好数を探すプログラムを作ってみました。リスト 1.1 が BASIC によるもの、リスト 1.2 はその C 言語版です。

次に示すのは、両者が友好数ペア (1184, 1210) を見つけるまでの実測値です。MSX-C は BASIC の 7 倍半のスピードでした。ちょっとしたものではないですか。

言語	実行時間
BASIC	2727秒
MSX-C	356秒

表 1.1 友好数ペア (1184, 1210) を見つけるまでの時間

次に、それならばということで、今度はこのプログラムを手作業でマシン語に書き直してみると、記録は 265 秒に縮まりました。さすがのコンパイラもこれにはかないません。

やっぱりそうか、コンパイラなんてマシン語を使えない人間のための二流の代用品なんだろうって？ いいえ、実はこのマシン語プログラミングに筆

```

100 DEFINT A-Z
110 FOR J=2 TO 10000
120   N=J: GOSUB 180: IF S<=J THEN 150
130   N=S: GOSUB 180: IF S<>J THEN 150
140   PRINT J;N
150 NEXT J
160 END
170 '
180 S=0
190 FOR I=N/2 TO 1 STEP -1
200   IF (N MOD I)=0 THEN S=S+I
210 NEXT I
220 RETURN

```

友好数とは、そのすべての約数(ただし自分自身は除く)の和が、互いに相手の数と等しくなるような数のペアのことです。

リスト 1.1 「友好数さがし」プログラム(BASIC 版)

```

#include <stdio.h>

main()
{
    int j, s, sum();

    for ( j = 2; j <= 10000; ++j ) {
        s = sum( j );
        if ( s > j && sum( s ) == j )
            printf( "%d %d\n", j, s );
    }
}

sum( n )
int n;
{
    int i, s;

    s = 0;
    for ( i = n/2; i >= 1; --i )
        if ( (n % i) == 0 ) s += i;
    return s;
}

```

リスト 1.2 「友好数さがし」プログラム(C 言語版)

者は1時間以上もかかっています。それに比べてCによるプログラミング時間はほんの数分でした。プログラムの作成から実行までをトータルで見れば、結局はコンパイラのほうが何倍も得だということになります。コンパイラは大いにものの役に立つのです。

言 語	合計時間(プログラム時間+実行時間)
BASIC 語	1 時間弱(約10分+2727秒)
MSX-C	10数分(約10分+356秒)
マシン語	1 時間以上(1 時間以上+265秒)

表 1.2 トータルで考えた所要時間

● MSX-C 対 MSX ベーしっ君

さて、MSX-C コンパイラを使うとマシン語なみのスピードが簡単に実現できると述べましたが、それは MSX-C に限らず他のどんなコンパイラでも同じことではあります。

たとえば前掲のリスト 1.1 を BASIC コンパイラ「MSX ベーしっ君」で実行してみたら、同じ友好数ベアを見つけるまでの時間は 258 秒でした。MSX-C より速いくらいです。げげッ、よく見たら筆者の書いたマシン語プログラムより速いではありませんか。

だったら、わざわざ MSX-C なんて持ち出してこないで、ベーしっ君でいいじゃないの、と考える方もいるでしょう。たしかに筆者もこのコンパイラにはよくお世話になります。とくに、20~30 行で書けるくらいの小さなプログラムは、ベーしっ君にかぎるとさえ思っています。

しかし、ある程度プログラムに本腰を入れて作るなら、やはり MSX-C を選ぶでしょう。なぜならプログラム全体の見通しのよさが全然違います。プログラムの規模が大きくなるほど、そのご利益も大きく現れてきます。これは「MSX-C 対ベーしっ君」というよりも C 言語と BASIC の持って生まれた性格の違いです。

● C はプログラミングしやすい「構造化言語」である

そもそも、どうすれば楽をしてプログラムが作れるか、この問題はコンピュータの誕生このかた多くの人々の考察のまとめでした。そして考え出されたのが、

①プログラムの構造化：

プログラムを部分品(モジュール)に分け、それを簡単な構造に従って組み立てる。

②データの構造化：

処理するデータをわかりやすい形にまとめて扱う。

という2つの方法でした。実際、この構造化方針に沿ってプログラムを作りはじめると、こんがらかったロジックが自然に避けられ、プログラミングは驚くほど楽になります。

C 言語はこの構造化プログラミングの考えを取り入れて作られています。ですから普通にプログラミングしていれば、いつのまにかプログラムやデータがモジュール化され、構造化されていきます。C 言語のプログラムの作りやすさの源はじつにこの点にあるわけです。それにひきかえ BASIC では、かなり意識的に努力しないと構造化プログラミングはできません。

このあたり、今のところはピンとこないかもしれません。またいくら文章で説明されても、たぶんなかなか理解できないことでしょう。しかし実際に MSX-C でプログラムを作りはじめれば、C のパワーがきつと実感できると思います。

●ライバルは Pascal か

どうも話がぐだぐだと長くなりました。簡単にいえば「コンパイラ型の構造化言語」は非常によろしいということです。だから、BASIC でもマシン語でもペーしっ君でもなく、MSX-C が一番なのです。そうではありませんか、皆さん！

しかし世の中油断できないもので、MSX-C がそう主張すると今度は「俺だって同じだ」と別のプログラミング言語が名のりをあげます。そのライバルの名は Pascal です。

同じコンパイラ型の構造化言語とはいえ、かたやCは実践派から、かたやPascalは理論派から出発した、かなり性格の異なるプログラミング言語です。ところが出会ってみれば、似たもの同士のソックリさんというわけでした。

あえて違いを探すなら、Cはプログラムの「作りやすさ」を目指し、Pascalはプログラムの「わかりやすさ」が第一だというところでしょう。たとえば、さきほどのリスト 1.2 の一部を Pascal で書き直すとリスト 1.3 のようになります。リスト 1.2 と比べてみれば、Cはプログラムの簡潔さを重視したリスト、Pascalは文書性を重視したリストになっていることがわかれると思います。

```
function sum( n : integer ) : integer;
var i, s : integer;
begin
  s := 0;
  for i := (n div 2) downto 1 do
    if (n mod i) = 0 then
      s := s + i;
  sum := s
end;
```

リスト 1.3 文書性は高いが簡潔さには欠ける Pascal のプログラム

● MSX-C はマシン語の力を取り入れやすい

C と Pascal の性格の違いは、このようなプログラムの字づら以外にも、いろいろな面で現われてきます。なかでもマシン語の扱いに関しては両者の方針にはだいぶ差があります。

まず Pascal のほうは、マシン語などという下々の言語とはいっさい関係を持たないことを前提にしています。せっかくコンパイラの導入によってプログラムがわかりやすくなったのに、そこにマシン語がしゃしゃりでてきては、もとのモクアミだという考えなのでしょう。

ところがCはマシン語を敬遠するどころか、これほどマシン語との相性がよい言語もありません。ダイヤモンドに日がくすみ、高級言語のたましいを捨てたな、という声がどこからか聞こえてきそうです。

しかしハードウェアの機能を目いっぱい引き出すには、マシン語との共存が不可欠なのです。とくに MSX というコンピュータは、VDP によるグラフィックスや、サウンドジェネレータ、マウスなど、面白い周辺装置をいろいろぶらさげています。このようなマシンをいじるには、MSX-C とマシン語の組み合わせこそ黄金のコンビネーションといってよいでしょう。下巻では、その具体例も紹介しますので期待しててください。

● MSX-C はシステムプログラミングの楽しさを味わえる

最後にもう 1 つ。MSX-C の作り出すプログラムには、UNIX 流のリダイレクション機能やパイプ機能が自動的に付加されます(4 章で解説)。これは、MSX-DOS TOOLS に代表される「ちょっといいツール」がガンガン作れるということです。

ただ、ここで「だから何？ ああいうののどこが いいの？」と聞かれると困ってしまいます。こういうレスポンスを返す人は、便利なツールというものの有用性を認めていないのでしょうね。

まあ、その気持ちはわからなくもありません。ツールのたぐいはプログラマにとって最も役に立つのですが、じゃあそれで何をプログラミングするかといえば、またまた別のツールを作ってみたりする人間が多い。風は風を呼び、ツールはツールを呼んで、結局なんのためのプログラムか、ということになります。

しかし筆者はあえていいます。このような「プログラムのためのプログラミング」は、実用性はともかく非常に楽しいものです。いや、そういうと語弊があるかもしれないので弁解しておきますが、もちろん C 言語は実用性だって十分です。でもそれ以上に「プログラマの知的オモチャ」として面白いんだからしょうがありません。弁解になってないか。でも、この楽しさが味わえるだけでも MSX-C には大きな存在価値があると、ここに断言してしましましょう。

● 結論：だから MSX-C

というわけで、ここまで述べてきた話を整理してみると、「MSX-C を使う」とい理由」は、次の 4 つになります。

- ①コンパイラであるため、手軽な割りに高速性を発揮できる
- ②構造化言語であるため、大きなプログラムも作りやすい
- ③マシン語との相性がよいため、MSX のハードウェアを活用しやすい
- ④理屈はとにかく、システムプログラミングの楽しさが味わえる

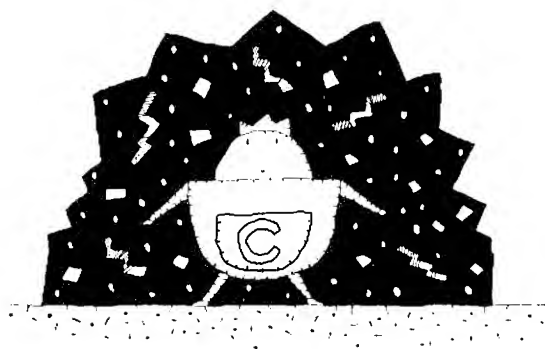
以上を総括すれば、MSX-C は「一歩先を目指すプログラミング言語」といえるのではないかと思います。より速く、より大きく、よりマニアックなプログラム、それが MSX-C のターゲットです。

BASIC やマシン語に満足している方には、MSX-C は無用のプログラミング言語かもしれません。しかし、いままでの自分のプログラムに満足できない方、ぜひ MSX-C を使ってみてください。新しいプログラミングスタイルに、目が開かれる思いをすること請け合いです。

さて次の章では、MSX-C を使用する前の準備として、プログラミング環境の整備を行います。MSX のスイッチを入れる用意をしてお待ちください。

2章

プログラミングの 環境を整える



MSX-Cを手にして、初めてDIRコマンドでディスクの中身を見ると、たいていの人が驚きます。こ、このファイルの多さはなんだあっ！

実はMSX-Cのシステムディスクの中には、MSX-Cコンパイラ本体に加えて、入門・中級者向けのサンプルプログラム、上級者向けのライブラリソースなどが混然一体となっているのです。

MSX-Cを使用するには、まず、このディスクから必要なファイルだけを抜き出し、若干のマシン語開発ツールと一緒にまとめて、プログラミング用のワークディスクを作ることから始めなくてはなりません。いわゆる「インストール」作業です。

本章では、このMSX-Cのインストールの方法を最初から順を追って説明していきます。これには結構時間もかかりますが、気合を入れて一気に片付けてしまいましょう。

21

これだけそろえば準備 OK

まず必要なハードウェアとソフトウェアを準備しましょう。あとからアレがない、コレがないとあわてないように、以下のものを全部机の上に並べてから作業を始めてください。備えあればうれしいな、という金言もあることですしね。

・ハードウェア

- ① MSX 本体
- ② ディスクドライブ(1 台以上)
- ③ ディスプレイ

・ソフトウェア

- ① MSX-C コンパイラ
- ② MSX-DOS TOOLS

・MSX-C インストール用のディスク

- ① 空ディスク(バックアップ用を含めて 2 枚あるとよい)
- ② ディスクラベル

● MSX 本体

MSX-C には、MSX-DOS1(以下 DOS1 と略)、あるいは MSX-DOS2(以下 DOS2 と略)の動作する、次のいずれかのマシンが必要です。

- | | | |
|-------|----|-------------------------|
| MSX1 | —— | 64K バイト以上のメイン RAM を持つもの |
| MSX2 | —— | どの機種でも可 |
| MSX2+ | —— | どの機種でも可 |

● ディスクドライブ

ディスクドライブは、本体内蔵型でも外付け型でも1台あれば十分です。2台あるとインストール作業には便利ですが、実際にMSX-Cを使い始めると、どちらでも大差ありません。

● ディスプレイ

ディスプレイに関してぜいたくはいいません。家庭用TVで結構です。ただし、MSX2あるいはMSX2+を使用していて、なおかつ今後C言語をバリバリ使おうという心づもりがあるなら、1行80字表示が可能な専用モニターのほうがよいでしょう。そのほうが見やすくカッコよいプログラムが書けます。

● MSX-C コンパイラ

MSX-C コンパイラには、現在のところ、Ver.1.1とVer.1.2の2種類のバージョンが存在しています。両者はそれぞれDOS1用のコンパイラとDOS2用のコンパイラという性格を持っていますが、Ver.1.1はDOS2で動作させることも可能です(ただしその場合、階層化ディレクトリをはじめとするDOS2特有の機能は、ほとんど活かせません)。

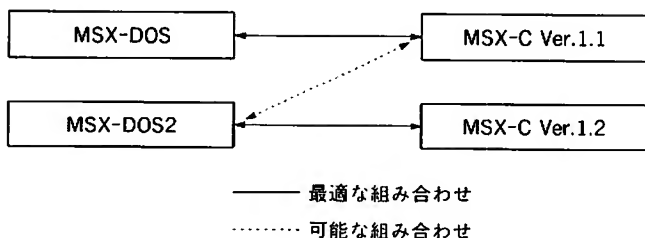


図 2.1 MSX-C コンパイラとMSX-DOSのバージョン対応関係

本書はどちらのバージョンのコンパイラをお使いの方でも読めるように書かれていますが、コンパイル操作などの実例を示す場合は、DOS1+Ver.1.1の環境下での実行画面を掲載しました。

● MSX-DOS TOOLS

MSX-C を利用するためには、MSX-C コンパイラ自体に加え、以下のプログラムツールが別途に必要です。

M80 アセンブラ

L80 リンクローダ

LIB80 ライブラリマネージャ

テキストエディタ

これらをそろえるには、すべてが1枚のディスクに収められた「MSX-DOS TOOLS」(以下 TOOLS と略)を購入するのが一番でとりばやいでしょう。本章でも、この TOOLS をお持ちになっていることを前提に、インストールの方法を説明していきます。

TOOLS には、このほかにもプログラミングの助けとなる便利なツールがいろいろとそろっていますから、ぜひ手に入れることをおすすめします。

● インストール用ディスクについて

必要なツール一式をそろえ、コンパイルの環境を整えるためには、やはりデータ容量の大きい2DDタイプのフロッピーディスクを使いたいものです。それでも1DDのフロッピーディスク1枚で間に合わないことはありません。必要なファイルをすべて1DDのディスクに集めても、百数十Kバイト程度の余地は残ります(プログラムを作っていくうちに、これくらいアツというまになくなりますけど)。

どちらにせよ、このディスクとは今後とも長い付き合いをすることになります。できればまっさらの新品を用意してください。そしてラベルに「C言語学習用」と大書して、フロッピーディスクの上に貼り付けておきましょう。さあヤルぞっと心も引き締まります。

2 インストールの手順

必要な機材はすべて準備できましたか？ そうしたら、以下の手順でワークディスクを作成します。

なお、DOS とコンパイラのバージョンの組み合わせ方によって、作成するディスクの内容構成に若干の差がありますので、説明は3つに分けておきました。みなさんのお手元のシステムに合わせて、必要なところだけお読みください。

■ DOS1 のためのインストール手順

DOS1+Ver.1.1 のシステムを使っている方は、以下の手順でインストールを行います。

① フォーマット済みのディスクを用意する

フォーマット作業はかならずDOS1で行ってください。異なるDOS (DOS2やMS-DOS)でフォーマットしたディスクを使うと、せっかくDOS1用の「MSXDOS.SYS」と「COMMAND.COM」をコピーしてもシステムは起動しません。ブートセクタに記録されるブートプログラムの内容が違って来るからです。

② TOOLS のディスクから必要なファイルをコピーする

TOOLS 中の必要なファイルを以下に示します。なお、MED 以外のエディタを利用されている方は、迷わずにそちらを使ってください。

MSXDOS.SYS	LIB80.COM
COMMAND.COM	MED.COM
M80.COM	MED.HLP
L80.COM	

③ MSX-C のマスターディスクから必要なファイルをコピーする

ここでは以下に示すファイルをコピーします。なお本書の後のほうでは、このほかのサンプルプログラムなども利用しますが、それらは必要になった時点で改めて指定させてもらいます。いまの時点では、この 13 個のファイルだけをコピーすれば結構です。

CF.COM	CK.REL
CG.COM	CLIB.REL
FPC.COM	CRUN.REL
LIB.TCO	CEND.REL
MX.COM	STDIO.H
AREL.BAT	BDOSFUNC.H
CREL.BAT	

MSXDOS	SYS	2432	85-08-23	9:29p
COMMAND	COM	6656	85-09-02	10:10p
M80	COM	20480	85-07-11	6:46p
L80	COM	10752	85-07-11	7:01p
LIB80	COM	4736	85-04-01	10:26p
MED	COM	15360	87-03-03	11:47p
MED	HLP	2196	87-02-09	11:27a
CF	COM	35072	87-08-12	12:00p
CG	COM	44800	87-09-29	1:07p
FPC	COM	28288	87-11-07	9:23p
LIB	TCO	2944	87-11-07	8:07p
MX	COM	11776	87-08-12	12:00p
AREL	BAT	24	87-08-12	12:00p
CREL	BAT	38	87-08-12	12:00p
CK	REL	128	87-08-12	12:00p
CLIB	REL	16000	87-11-07	8:07p
CRUN	REL	1664	87-08-12	12:00p
CEND	REL	128	87-08-12	12:00p
STDIO	H	3153	87-11-04	7:27p
BDOSFUNC	H	5122	87-08-12	12:00p
20 files		506880	bytes	free

図 2.2 インストールを終了したディスクの内容

④ 追補：VRAM ディスクについて

これはオマケですが、128K の VRAM を持つ機種(MSX2/2+)を使用している方は、さらに Appendix2 の VRAM ディスクをインストールすると、快適な MSX-C ライフをおくることができます。

MSX-DOS は基本的にグラフィックスを使わないため、必要な VRAM はたかだか 6K バイトですから、MSX2 なら 100K バイト以上の VRAM が手付かずの状態に残っています。VRAM ディスクは、この部分を仮想ディスクとして利用するものです。これはフロッピーディスクよりずっと高速で、とくに MSX-DOS の外部コマンドが瞬時に実行できる快感は一度味わうとやみつきになるでしょう。

VRAM ディスクを利用するには、Appendix2 に示した手順に従って必要な 4 つのファイルを作り、ワークディスクに追加してください。ただしプログラムは 16 進ダンプリストの形で掲載してあります。これを打ち込むのは、かなりホネかもしれません。

■ DOS2+Ver.1.1 のためのインストール手順

DOS2+Ver.1.1 のシステムを使っている方は、ここに述べる手順でインストールを行います。

① フォーマット済みのディスクを用意する

フォーマット作業はかならず DOS2 で行ってください。DOS1 でフォーマットしたディスクは、いったん DOS2 上で FIXDISK コマンドにかけて正しいフォーマットに更正してから使う必要があります(FIXDISK コマンドの使い方は DOS2 のマニュアルを参照のこと)。

MSX-DOS 以外(MS-DOS など)でフォーマットしたディスクは、ブートセクタに記録されるブートプログラムの内容が異なるため使用できません。

② システムファイルをコピーする

DOS2 に付属してくるシステムディスクから、以下に示す 2 つのシステムファイルをコピーします。

MSXDOS2.SYS
COMMAND2.COM

③ 起動時実行バッチファイルを作る

システム起動時に MSX-C の実行環境を整えるため、以下の2つのバッチファイルを作成します。

```
randisk 4064/d  
copy command2.com h:¥  
reboot %1
```

リスト 2.1 AUTOEXEC.BAT

```
set temp=h:¥  
set shell=h:¥command2.com  
set append=%1¥files  
path %1¥bin
```

リスト 2.2 REBOOT.BAT

④ サブディレクトリを作る

必要なファイルを置くための、以下の3つのサブディレクトリを作ります。

BIN (コマンドを置く)
FILES (読み出し専用ファイルを置く)
WORK (作業用)

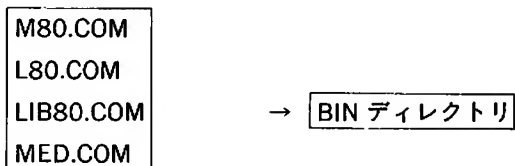
環境変数 PATH によって指定される BIN ディレクトリには、コマンドファイル(拡張子 COM や BAT を持つもの)を置きます。

環境変数 APPEND によって指定される FILES ディレクトリには、各種の読み出し専用ファイル(インクルードファイル、ライブラリファイル、ヘルプファイル)を置きます。

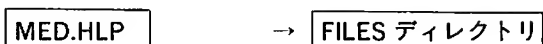
そして WORK ディレクトリで実際のコンパイル作業を行います。

⑤ TOOLS のディスクから必要なファイルをコピーする

以下のファイルを BIN ディレクトリの下にコピーします。

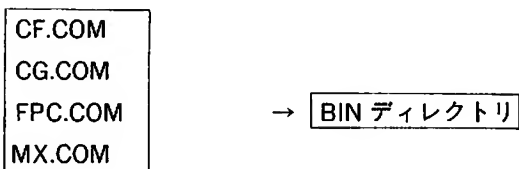


以下のファイルを FILES ディレクトリの下にコピーします。

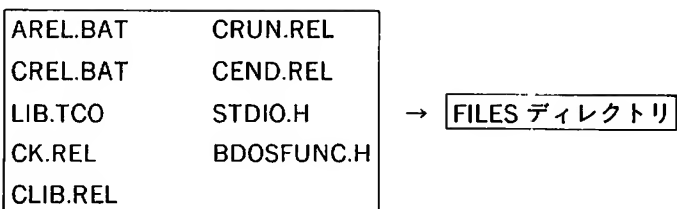


⑥ MSX-C のオリジナルディスクから必要なファイルをコピーする

以下のファイルを BIN ディレクトリの下にコピーします。



以下のファイルを FILES ディレクトリの下にコピーします。



「AREL.BAT」と「CREL.BAT」は拡張子 BAT を持っていますが、実際にはバッチファイルではなく MX コマンドの使うデータファイルです。こちらのディレクトリに置いて間違いではありません。

```
Volume in drive B: has no name
X-Directory of B:¥
```

```
MSXDOS2.SYS      4480
COMMAND2.COM     14976
AUTOEXEC.BAT      50
REBOOT.BAT       75
```

```
¥BIN
```

```
  M80.COM        20480
  L80.COM        10752
  LIB80.COM       4736
  MED.COM        15360
  CF.COM         35072
  CG.COM         44800
  FPC.COM        28288
  MX.COM         11776
```

```
¥FILES
```

```
  MED.HLP        2196
  AREL.BAT        24
  CREL.BAT        38
  LIB.TCO        2944
  CK.REL         128
  CLIB.REL       16000
  CRUN.REL       1664
  CEND.REL       128
  STUDIO.H       3153
  BDOSFUNC.H     5122
```

```
¥WORK
```

```
217K in 22 files  480K free
```

図 2.3 インストールを終了したディスクの内容を XDIR コマンドで表示

■ DOS2+Ver.1.2 のためのインストール手順

DOS2+Ver.1.2 のシステムを使っている方は、ここに述べる手順でインストールを行ってください。

① フォーマット済みのディスクを用意する

フォーマット作業はかならず DOS2 で行ってください。DOS1 でフォーマットしたディスクは、いったん DOS2 上で FIXDISK コマンドにかけて正しいフォーマットに更正してから使う必要があります (FIXDISK コマンドの使い方は DOS2 のマニュアルを参照のこと)。

MSX-DOS 以外 (MS-DOS など) でフォーマットしたディスクは、ブートセクタに記録されるブートプログラムの内容が異なるため使用できません。

② システムファイルをコピーする

DOS2 に付属してくるシステムディスクから、以下に示す 2 つのシステムファイルをコピーします。

```
MSXDOS2.SYS  
COMMAND2.COM
```

③ 起動時実行バッチファイルを作る

システム起動時に MSX-C の実行環境を整えるため、以下の 2 つのバッチファイルを作成します。

```
ramdisk 4064/d  
copy command2.com h:¥  
reboot %1
```

リスト 2.3 AUTOEXEC.BAT

```

set temp=h:¥
set shell=h:¥command2.com
set include=%i¥include
set append=%i¥lib
path %i¥bin

```

リスト 2.4 REBOOT.BAT

④ サブディレクトリを作る

必要なファイルを置くための、以下の4つのサブディレクトリを作ります。

BIN	(コマンドを置く)
INCLUDE	(インクルードファイルを置く)
LIB	(ライブラリファイルを置く)
WORK	(作業用)

環境変数 PATH によって指定される BIN ディレクトリには、コマンドファイル(拡張子 COM や BAT を持つもの)を置きます。

環境変数 INCLUDE によって指定される INCLUDE ディレクトリには、MSX-C のヘッダファイル(拡張子 H を持つもの)を置きます。

環境変数 APPEND によって指定される LIB ディレクトリには、ライブラリファイルの他、いくつかの読み出し専用ファイルを置きます。

そして WORK ディレクトリで実際のコンパイル作業を行います。

⑤ TOOLS のディスクから必要なファイルをコピーする

以下のファイルを BIN ディレクトリの下にコピーします。

```

M80.COM
L80.COM
LIB80.COM
MED.COM (DOS2 TOOLS の場合は KID.COM または AKID.COM)

```

MED を使う場合には以下のファイルを LIB ディレクトリの下にコピーします。

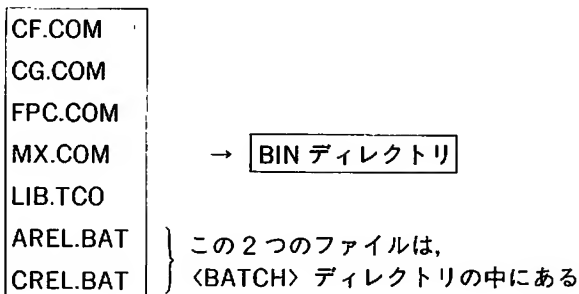
```

MED.HLP

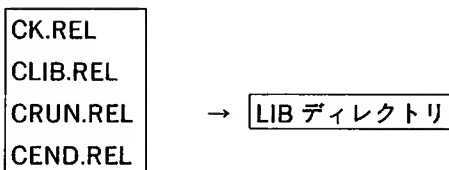
```

⑥ MSX-C のマスターディスクから必要なファイルをコピーする

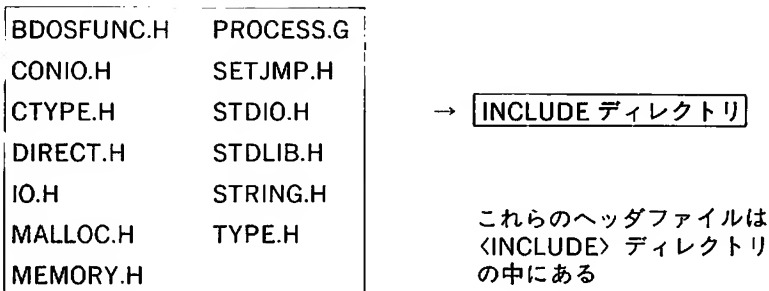
以下のファイルを BIN ディレクトリの下にコピーします。なお、「LIB.TCO」はコマンドファイルではなく、FPC コマンドが使うデータファイルですが、MSX-C コンパイラの Ver.1.2 では FPC と一緒にディレクトリに置くことになっています。また「AREL.BAT」と「CREL.BAT」も MX コマンドが使うデータファイルですが、MX と一緒にディレクトリに置くことになっています。



以下のファイルを LIB ディレクトリの下にコピーします。



以下のファイルを INCLUDE ディレクトリの下にコピーします。これはマスターディスクの INCLUDE ディレクトリの全内容です。



Volume in drive B: has no name
X-Directory of B:¥

MSXDOS2.SYS	4480
COMMAND2.COM	14976
AUTOEXEC.BAT	50
REBOOT.BAT	76
¥BIN	
M80.COM	20096
L80.COM	10752
LIB80.COM	4480
KID.COM	28928
AKID.COM	24576
CF.COM	34816
CG.COM	45056
FPC.COM	26368
MX.COM	10112
LIB.TCO	4168
AREL.BAT	23
CREL.BAT	36
¥INCLUDE	
BDOSFUNC.H	8473
CONIO.H	315
CTYPE.H	922
DIRECT.H	285
IO.H	792
MALLOC.H	420
MEMORY.H	259
PROCESS.H	289
SETJMP.H	331
STDIO.H	2517
STDLIB.H	306
STRING.H	424
TYPE.H	588
¥LIB	
CK.REL	128
CLIB.REL	17536
CRUN.REL	1664
CEND.REL	128
¥WORK	
258K in 33 files	432K free

図 2.4 インストールを終了したディスクの内容を XDIR コマンドで表示

■ 最後のチェック (全バージョン)

以上で MSX-C の実行に必要な全ファイルの用意ができました。念のため、このワークディスクで MSX-DOS が起動できるかどうか確認しましょう。リセットスイッチを押すか電源を切ってもういちど入れ直し、うまく MSX-DOS が起動すれば、胸をなでおろしてください。

そして最後に、もう 1 枚フォーマット済みのディスクを用意し、これまで作ったディスクの内容をすべてコピーしておきましょう。このバックアップディスクは、大切なプログラムやデータを保存しておくために使用します。

MSX-C でプログラミングしていると、BASIC とは比較にならないほど、しょっちゅうディスクがアクセスされます。そのため、いつディスクが劣化して重要なファイルが読めなくなるかわかりません。そのような万一の場合に備えるのが、ここで用意したバックアップディスクの役割です。保存に値すると思うプログラムなどは、入力し終えたらすかさずバックアップを取っておく習慣をつけると安心です。

さて、やっと MSX-C の利用体勢も整いました。昨今はやりの言葉でいえば、これでインフラストラクチャが完備したというわけです。次の章では、ここで作ったワークディスクを用いて、プログラムのコンパイルから実行までの方法を説明することにします。

3章

コンパイラの動作と その使い方



この章では、MSX-CコンパイラがCのソースプログラムをどうやってマシン語まで持っていくのか、その処理の流れを見ていただきます。

MSX-Cで我慢ならないのは、プログラムの実行まで何分も待たされるコンパイルのわずらわしさだ！ そんな不満を持つ方は、ここで紹介したコンパイル作業の内容を見てやってください。これじゃ時間がかかるのも無理はないと、少しはMSX-Cに同情する気が起きるかもしれません。

なお、理屈はともかく早くコンパイラを動かしてみたいんだという方は、とりあえず本章の最後の「CC.BATによるコンパイル」の項だけ目を通して、そのまま先へ進んでも結構です。

3 1 ソースプログラムの準備

ここでの第一の目的はコンパイル操作の練習です。そのためには練習台となる適当なCのプログラムが1つ必要です。

そんなプログラムがどこかに都合よくころがっていないか、と探してみたら、ありました、ありました。MSX-Cのオリジナルディスクは、サンプルプログラムの宝庫です。本章では、この中の「WC.C」を使って、コンパイルの練習をすることにします。

まずは、このサンプルプログラムをMSX-Cのオリジナルディスクからコピーしてください(図3.1)。

```
A>copy b:wc.c ☒ .....ドライブBのWC.CをドライブAにコピー
1 file copied
A>dir wc ☒ .....正しくコピーできたか確認
WC      C      2223 87-08-12 12:00p
1 file  503808 bytes free
```

注：MSX-C Ver.1.2では、サンプルプログラムWC.Cは<SAMPLE>というディレクトリに収められています

図 3.1 WC.Cのコピー

このプログラムはファイルに含まれる行・文字・単語の数を調べるためのもので、WC という名前はワードカウント (WordCount) の頭文字をとってつけられています(WC といってもトイレではないんですね)。

ただし、本章ではまだプログラムの作り方を説明するわけではありませんから、プログラムの内容については理解できなくともかまいません。要するにバグがなくエラーを起こさないでコンパイルできれば、どんなプログラムでもよいのです。

■ MSX-C の 4 段階活用

さて BASIC ならば、こうしてプログラムの用意ができたらすぐにも実行させられるのですが、C の場合はそう簡単にいきません。いったんこれをマシン語プログラムに変換しなければなりません。

このマシン語への変換作業のことをコンパイルといいます。そして、コンパイルされる前の C 言語のプログラムを一般にソースプログラム、結果として作成されるマシン語プログラムをオブジェクトプログラムと呼びます。なお、C のソースプログラムには、「WC.C」のように必ず「.C」という拡張子を付ける約束になっています。

ところで一口にコンパイルといっても、MSX-C コンパイラは実は 1 個の独立したプログラムではありません。ソースプログラムを最終的なマシン語オブジェクトに変換するには、図 3.2 の 4 つのコマンドを順に実行する必要があります。

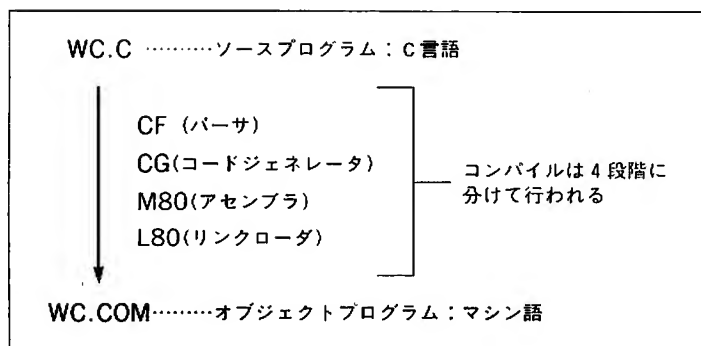


図 3.2 MSX-C の 4 段階の処理

通常のコンパイルでは、あとでも述べるように、この 4 つのコマンドはバッチ処理にまとめて実行されます。しかしここでは各コマンドの役割を理解するため、1 つずつ順番にその動作を追ってみましょう。

CF の動作

コンパイルの最初に実行するコマンドが CF です。CF はソースプログラムを解析して T コードと呼ばれる記号に変換し、その結果を「.TCO」という拡張子を持つファイルに書き出します。

● CF を実行する

それでは、用意した「WC.C」を図 3.3 のように CF でコンパイルしてください。このときファイル名に拡張子は付けず、ただ WC とだけ指定します。

ただし、これは MSX-C Ver.1.1 だけの制約で、Ver.1.2 からはファイル名に「.C」という拡張子を含めてもよいことになりました(拡張子を付けなければ、Ver.1.1 と同様に「.C」が指定されたものとみなします)。実は一般の C はみな拡張子も含めてファイルを指定する方法をとっていて、拡張子を省略する MSX-C Ver.1.1 のようなコンパイラは、ごくごく少数派なのです。

```
A>cf wc [C] .....CFの実行(ファイル名に拡張子は付けない)
MSX C ver 1.10P (parser)
Copyright (C) 1987 by ASCII Corporation
complete
```

図 3.3 CF の実行

コマンドの実行が終わると、「WC.TCO」という T コードファイルが作られているはずです。これを DIR コマンドで確認してください(図 3.4)。

```
A>dir wc [C]
WC      C      2223 87-08-12 12:00p
WC      TCO    3200 89-04-10 1:00p .....Tコードファイル
      2 files  499712 bytes free
```

図 3.4 T コードファイルの確認

● CFは何をする人ぞ

ここで、CFがどのような仕事をしているのか簡単に述べておきます。

CFはまず「#include」や「#pragma」など「#」ではじまる特殊命令を実行します。その機能はさまざまですが、実際のコンパイルの前処理という意味で、これらの実行はプリプロセス(pre-process)と呼ばれます(詳しくは各命令の説明を参照)。

次にCFは数字や名前を識別し、たとえば「ab = cde + fg;」という文なら、「ab」「=」「cde」「+」「fg」「;」という6語に分解します。この作業のことを字句解析、横文字でレキシカルアナライズ(lexical-analyze)といいます。

そして最後に、得られた単語の並び方から、この文は代入文であるとか、ここからここまでがループだとか、プログラムの意味の解釈を行います。これを構文解析、あるいはパース(parse)といいます。

つまり、CFは単に「パーサ」と呼ばれていますが、実際にはプリプロセッサとレキシカルアナライザとパーサ、この三役を務めているわけです。

● Tコードをのぞいてみる

CFはこうして人間の書いたソースプログラムを解析し、コンピュータの理解しやすいTコード形式に変換して、コンパイラの次段に渡します。

このTコードはもとのソースプログラムより機械的に処理しやすいため、コンパイラだけではなく、MX(モジュールエクストラクタ)やFPC(ファンクションパラメータチェッカ)など各種ユーティリティプログラムでも利用され、MSX-Cにとっては欠かせない存在となっています。

そこでTコードの中身はどうなっておるのだろうかとTYPEコマンドで表示してみると、すべての変数が1から始まる通し番号で管理されていたり、数式が逆ポーランド記法に変わっていたり、なかなか感心させられてしまいました。Tコードの構造は公表されていないので詳しいところは不明でしたが、ヒマと興味のある方は解読してみてください。

CG の動作

CF の次に実行するコマンドは CG です。CG は T コードファイルを解析して、Z80 のアセンブリ言語に変換し、その結果を「.MAC」という拡張子を持つファイルに書き出します。

● CG を実行する

それではさきほど作られた「WC.TCO」を CG に与え、コンパイルの第 2 ステップを実行してください。このとき図 3.5 のように CG コマンドに与えるファイル名には、やはり拡張子は付けません (Ver.1.2 では拡張子を付けてもかまいません)。

```

A>cg wc ..... Ver.1.1ではファイル名に拡張子は付けない
               Ver.1.2では付けてもよい
MSX C ver 1.10r (code generator)
Copyright (C) 1987 by ASCII Corporation
zerolong .....;
inlong .....;
putlong .....;
count .....;
main .....;
complete .....;
  
```

関数名 文の処理開始 最適化開始

図 3.5 CG の実行

CG コマンドは実行中いろいろな途中経過を報告してくれます。まず関数の処理を始めるとその名を表示し、その中で文を処理するごとにピリオドを表示します。コンパイルの進行状況がいつでも見えているおかげで、まだ終わんないのかコイツめ、というイライラも少しは解消するでしょう。

面白いのは、プログラムの最適化を行うごとにコロが表示されることです。1個のコロンが出てくると、思わず MSX-C ガンバレと声をかけたくなります (そんなの筆者だけですかね)。

そして最後に complete と表示されれば、CG は無事終了です。

ここでは「WC.MAC」というファイルが作られているはずです。やはり DIR コマンドで確認してください(図 3.6)。

```
A>dir wc
WC      C      2223 87-08-12 12:00p
WC      TCO    3200 89-04-10  1:00p
WC      MAC    5376 89-04-10  1:02p .....アセンブリ言語プログラム
          3 files 493568 bytes free          が作られた
```

図 3.6 WC.MAC の確認

● CG の作るマシン語プログラム

この段階で作られるファイルは、アセンブリ言語で書かれたマシン語ソースプログラムです。Z80 のマシン語をご存じの方は「WC.MAC」を TYPE コマンドで表示してみてください。見覚えのあるマシン語命令が並んでいるでしょう(図 3.7)。マシン語プログラミングに慣れているなら、このプログラムの無駄な部分に手をを入れて改善することもできます。

```
zerolo:
        push    hl
        inc     hl
        inc     hl
        ld      (hl),0
```

図 3.7 WC.MAC の一部を見る

ただし、「WC.MAC」に書き出されるのは、あくまでも元のソースプログラムに対応する部分だけです。文字列表示などの基本的なルーチンは、後述する「CLIB.REL」などのライブラリファイル中にリロケートブルモジュールとして記録されていて、ここでは出力されません。

M80 の動作

CG の次に実行するコマンドが M80 です。M80 はマシン語ソースファイルをアセンブルし、「.REL」という拡張子を持つファイルを作ります。

● M80 を実行する

それでは図 3.8 のように「WC.MAC」をアセンブルしてください。このとき M80 コマンドに特有の約束事として、ファイル名の前に「=」を付けることと、ファイル名の後ろに「/Z」を付けることを忘れてはいけません。なぜか？ という質問は発さないでください。M80 というコマンドは、こう使うと決められているのです。

本書の範囲を超えるため、ここでは M80 に関する説明はひかえますが、MSX-DOS TOOLS のマニュアルには、M80 の使い方の説明が詳しく解説されています。それを読めば M80 の使い方もよくわかる(のではない)でしょう(か)。

```
A>m80 =wc/z  [F5] .....M80の実行
No Fatal error(s)
```

図 3.8 M80 の実行

実行が終わると「WC.REL」というファイルが作られます。やはり DIR コマンドで確認してください(図 3.9)。

```
A>dir wc  [F5]
WC      C      2223 87-08-12 12:00p
WC      TCD    3200 89-04-10  1:00p
WC      MAC    5376 89-04-10  1:02p
WC      REL    1280 89-04-10  1:05p .....リロケートブルモジュール
      4 files  491520 bytes free
```

図 3.9 WC.REL の確認

●リロケートブルモジュールについて

M80の作るファイルはリロケートブルモジュールと呼ばれ、複数のモジュール同士を組み合わせて簡単にマシン語プログラムが作れるように工夫されています。ですから、いろいろ便利なサブルーチンをリロケートブルモジュールの形でたくさん用意しておけば、あとはそれを積木のように組み立てるだけでプログラムが作れてしまいます。

MSX-Cには、入出力やファイル操作などの基本的なサブルーチンが、このリロケートブルモジュールの形でたくさん用意されています。いま作られた「WC.REL」も、それらのモジュールと一緒にまとめられて最終的なマシン語プログラムになっていくわけです。

リロケートブルモジュールというものは、MSX-Cが開発される前からマシン語プログラムの開発に使われていたのですが、こいつがあればこそ、MSX-Cもこの世に生まれ出づることができたのでしょう。リロケートブルモジュールとは、それくらい重要なアイテムなのです。

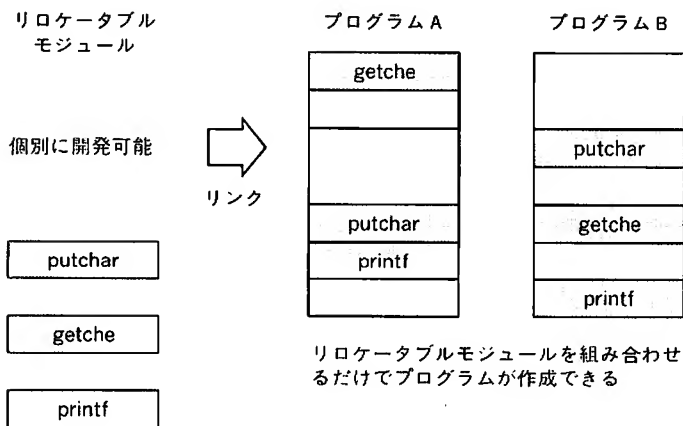


図 3.10 モジュールごとのプログラム開発

ところでリロケートブルモジュールは T コードファイルやアセンブリ言語ファイルとは違い、TYPE コマンドで内容を見ることができません。また、データがビット単位で詰め込まれているため、1 バイトごとに 16 進表示して

みても、何が記録されているのか判別できません。そのためにロケートブルモジュールの中身を知るには、LIB80 コマンドを使わなければなりません。その具体的な方法は下巻で説明しましょう。

■ L80 の動作

いよいよ最終段階です。ここでは L80 を用いてロケートブルモジュールをリンクし、実行可能なマシン語プログラムを作ります。完成したプログラムは「.COM」の拡張子を持つファイルに書き込まれ、MSX-DOS の外部コマンドになります。

● L80 を実行する

L80 の実行方法は図 3.11 に示すとおりです。指定するパラメータが多いので、間違いのないように入力してください。

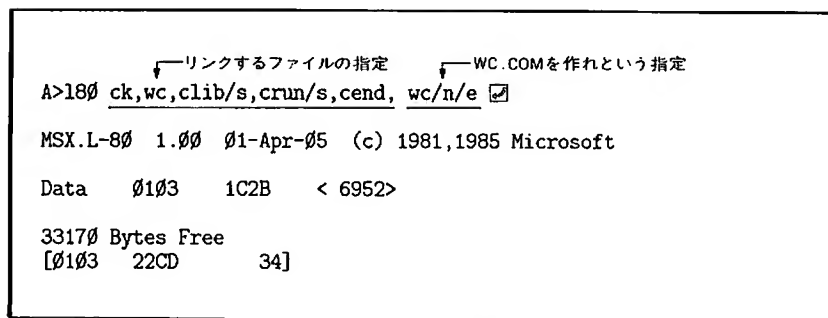


図 3.11 L80 の実行

ここで L80 に与えているパラメータは、「CK.REL」「WC.REL」「CLIB.REL」「CRUN.REL」「CEND.REL」の 5 ファイルをリンクし、「WC.COM」を作れということを意味しています。

これらのファイルの中で、「CLIB.REL」と「CRUN.REL」の 2 つは複数のモジュールをまとめたライブラリファイルです。L80 を実行するとき、ファイル名の後ろに「/S」スイッチを指定しておくと、そのファイルが必要となき

に限りリンクさせるという芸当が可能です。さらにそれがライブラリファイルの場合は、そこから必要なモジュールだけを自動的に選び出してくれます。いやあ、L80ってホントに便利なツールですね。

L80 によるリンクが終了すると、「WC.COM」が作られています。やはりこれも DIR コマンドで確認してください(図 3.12)。

```
A>dir wc [ ] .....結果の確認
WC      C      2223 87-08-12 12:00p
WC      TCO    3200 89-04-10 1:00p
WC      MAC    5376 89-04-10 1:02p
WC      REL    1280 89-04-10 1:05p
WC      COM    8704 89-04-10 1:10p .....WCコマンド完成
          5 files  482304 bytes free
```

図 3.12 WC.COM の確認

さあこれでコンパイルは終了、WC コマンドが完成しました。前にも述べたとおり、このコマンドはファイルの中の行数・単語数・文字数をカウントするものです。ために「WC.C」自身の内容を調べてみましょう(図 3.13)。

```
A>wc wc.c [ ]
      145      283      2077 .....WC.Cには145行、283単語、2077文字
                               が含まれていることを表す
```

図 3.13 完成した WC コマンドの実行

やった成功だ！ というわけで、手間をかけたわりに結果はささやかなものでしたが、どうやらコンパイルは成功したようです。

●モジュールたちのプロフィール

上の作業で、L80 はターゲットの「WC.REL」以外に 4 つのファイルをリンクしました。これらはそれぞれ以下に述べるような役割を持っています。この 4 つは、どんなプログラムをコンパイルするときでも、かならずリンクしてやる必要があります。

① CK.REL —— カーネル起動プログラム

カーネルとは、コマンドに与えられたパラメータを受け取ったり、リダイレクトやパイプ処理の準備を整えて、プログラム本体(main関数)を実行するルーチンのことです。そして、そのカーネルを起動するのが「CK.REL」です。なおカーネル自体は次に述べる「CLIB.REL」の中に格納されていて、そのリンク時に取り込まれます。

② CLIB.REL —— 基本関数ライブラリ

これは各種の基本的な関数のモジュールを集めたライブラリです。リンクするときに「/S」という指定を付けておけば、L80はこの中から必要なモジュールだけを選択します。

たとえば「CK.REL」と「WC.REL」をリンクすると、その段階でカーネルや文字列表示ルーチンが必要とされていることがわかります。そこでL80は、それらを含んだモジュールを「CLIB.REL」から抜き出すというわけです。

③ CRUN.REL —— ランタイムルーチン・ライブラリ

これは乗除算や数値の比較など、プログラムの部品となる小さなサブルーチン(いわゆるランタイムルーチン)を集めたライブラリです。「CLIB.REL」と同様に、この中の必要モジュールだけがリンクされます。

「CRUN.REL」と「CLIB.REL」は、1つのライブラリにまとめても問題はないのですが、一人前の関数とランタイムルーチンのケジメを付けるため、MSX-Cでは別々にまとめられているようです。

④ CEND.REL —— プログラムエンドの目印

このモジュールは他のすべてのモジュールの後ろにリンクされ、プログラムの最終アドレスの目印になります。作業用のメモリ領域を必要とするプログラムなどでは、この目印を利用して空きメモリの先頭アドレスを知ります。

3 2 CCコマンドによる コンパイル

以上、MSX-Cによるコンパイルの手順を紹介してきました。しかしコンパイルのたびにこのような作業を進めるのは、いくらなんでも手間がかかりすぎるため、通常はこれらをバッチコマンドにまとめて実行します。ついでにバッチコマンドの最後で、コンパイラが作った中間生成ファイルを消してしまえば、ディスクもスッキリするでしょう。

実はMSX-Cのオリジナルディスクには、そのための「C.BAT」というバッチファイルがすでに存在しているのですが、残念ながら「C.BAT」のコンパイル手順にはここまでの説明と少し異なる部分があります。そこで本書では新しく「CC.BAT」というバッチファイルを作ることになりました。

■ 「CC.BAT」の作成

リスト 3.1 が「CC.BAT」の内容です。スクリーンエディタなどを使って入力してください。

```
cf %1
cg %1
m80 =%1/z
del %1.tco
l80 ck,%1,clib/s,crun/s,cend, %1/n/e
del %1.mac
del %1.rel
```

リスト 3.1 「CC.BAT」 —— コンパイラの元締め役

この中で、L80 コマンドの実行前に「del %1.tco」を行って T コードファイルを消去しているのは、図 3.14 に示すように、ディレクトリを表示したと

きソースファイルの直後にコマンドファイルが来るようにするための工夫です。A 型人間はこういう部分についてこだわります。

① T コードを消去してからCOMファイルを作った場合のディレクトリ

⋮	⋮	⋮	⋮	
WC.C	WC.C	WC.C	WC.C	すぐあとに 作られる
WC.TCO	消 去	WC.COM	WC.COM	
WC.MAC	WC.MAC	WC.MAC	消 去	
WC.REL	WC.REL	WC.REL	消 去	

② T コードを消去しないでCOMファイルを作った場合のディレクトリ

⋮	⋮	⋮	
WC.C	WC.C	WC.C	3 つ先に離れてしまう
WC.TCO	WC.TCO	消 去	
WC.MAC	WC.MAC	消 去	
WC.REL	WC.REL	消 去	
	WC.COM	WC.COM	

図 3.14 リンクの前に T コードファイルを消去する理由

また、RAM ディスクや VRAM ディスクが利用できるならば、そこに中間ファイルを作るとコンパイルのスピードが少し速くなります。たとえば RAM ディスクが H ドライブに設定されている場合は、リスト 3.2 に示すような「CC.BAT」を作成してください。

```
cf -oH %1
cg -oH H:%1
m80 =H:%1/z
180 ck,H:%1,clib/s,crun/s,cend, %1/n/e
del H:%1.tco
del H:%1.mac
del H:%1.rel
```

リスト 3.2 RAM ディスク利用時の CC.BAT

■ 再び「WC.C」のコンパイル

それではCCコマンドを使って、再び「WC.C」をコンパイルしてみましょう。

CCコマンドを使用するには、次のように拡張子を除いたファイル名(ここではWC)をCCの後ろに指定します。

```
A>cc wc
```

たったこれだけで、あとは「CC.BAT」がすべての作業を肩代りしてくれるのですから、バッチコマンドとは偉大なものです。ある友人の曰く、「ひとつのファイルはすべてを調べ、ひとつのファイルはすべてを見つけ、ひとつのファイルはすべてを捕えて、くらやみのなかにつなぎとめる。コマンド横たわるバッチ処理のなかに……」だとか。

CCの実行が終了したら、「WC.COM」が作られ、中間ファイルは消されていること、およびWCコマンドが前と同様に動作することを確認してください(図3.15)。

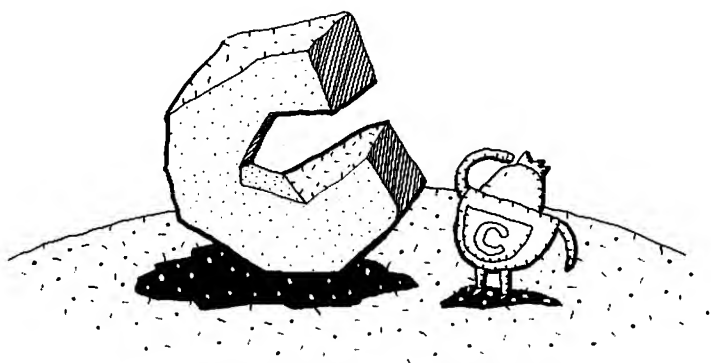
```
A>dir wc [ ] ..... CC.BAT で作られたWC.COMを確認する
WC      C      2223 87-08-12 12:00p
WC      COM    8704 89-04-10  1:15p ..... WCコマンドだけ残り、中間
      2 files  493568 bytes free          ファイルは消された
A>wc cc.bat [ ]
      7      17      112 ..... WCコマンドの動作を確認
```

図 3.15 CC.BAT の動作確認

コンパイルは成功しましたか？ うまく動作しなかった場合は「CC.BAT」の内容をもう1度チェックしてください。次の章からは、このCCコマンドを使ってプログラムのコンパイルを行っていきます。

4章

これがCの プログラムだ



みなさんが初めてCのプログラムを見たとき、どんな感想をお持ちになりましたか。筆者の場合は、ムムッこいつはBASICとはだいぶ違うな、ということでした。まず見た目からして、プログラムは小文字で書かれているわ、行番号は存在しないわ、カッコのたぐいは妙に多いわ。

さらに、プログラムを実行するときの形態が、これまたBASICとCではかなり違います。BASICのプログラムは、まずRUNして、それからデータを入力して、というような「対話的」な環境で実行されるものでしたが、Cではそんな悠長な姿勢はハヤリません。コマンドラインで指定したパラメータをもって、ドカッとプログラムを実行してしまいます(前章で作ったWCコマンドがいい例です)。

このような言語を使うからには、BASICのときとはいささか発想も転換する必要があります。本章では、いくつかのサンプルプログラムを通して、まずはこういったC特有のプログラミングスタイルに慣れていただくことにしましょう。

41 Cのプログラムに触れてみる

さて前章で「CC.BAT」の用意も整い、Cで書かれたプログラムは自由にコンパイルできるようになりました。こうしてコンパイラが作り出したオブジェクトプログラムは、次のような使い方ができる点に大きな特徴があります。

- ①起動時にパラメータを指定できる。
- ②入出力のリダイレクションが行える。
- ③プログラムをパイプで何段も重ねられる。

そこで、まずは典型的なCのプログラムを例にとりて、その使い方を体験してみることにしましょう。

■ プログラムの入力から実行まで

というわけで、また練習台となる適当なプログラムが必要になるのですが、MSX-Cのマスターディスクに収められているサンプルプログラムというやつは、前章で利用した「WC.C」も、それ以外のサンプルも、どれも出力結果がジミで、実行してもあまり面白味がありません。

そのため、ここでは入力の実習も兼ね、プログラムが簡単で、ちょっと見栄えのするプログラム「TRIANGLE.C」を用意しました。次のページにそのリストを示します(図4.1)。これを打ち込んでサンプルプログラムとして利用しましょう。

```

0: #include <stdio.h>
1:
2: putline(ch, n)
3: char  ch;
4: int    n;
5: {
6:     while (--n >= 0)
7:         putchar(ch);
8: }
9:
10: main(argc, argv)
11: int  argc;
12: char *argv[ ];
13: {
14:     int  i;
15:     int  size, spc, len;
16:
17:     if (argc < 2)
18:         size = 5;
19:     else
20:         size = atoi(argv[1]);
21:
22:     len = 1;
23:     spc = size;
24:     for (i = 1; i <= size; ++i) {
25:         putline(' ', spc);
26:         putline('+', len);
27:         putline(' ', spc-1);
28:         putchar('\n');
29:         len += 2;
30:         --spc;
31:     }
32: }

```

図 4.1 サンプルプログラム "TRIANGLE.C"

BASIC では、プログラムの入力も実行も、すべて BASIC という 1 つの環境の中でまかなうことができました。それに対して C では、プログラムの入力・コンパイル・実行の 3 つは、それぞれまったく別の操作になります。

●プログラムの入力

まずプログラムの入力には、テキストエディタと呼ばれる文字入力の専用ツールを使います。その入力方法は個々のエディタによっても異なりますが、

たとえばMSX-DOS TOOLSに収められているMEDというエディタを使う場合には、次のような手順をとります。

① テキストエディタ MED の起動

MSX-DOSのコマンドラインから次のようにMEDを起動します。ここで指定している「TRIANGLE.C」は、作成したいファイルの名前です。すでに同じ名前のファイルが存在していれば、その内容が読み込まれてからエディタが起動します。

```
A>med triangle.c
```

② プログラムの入力

MEDを起動させたら、リストを見ながら、ひたすらプログラムを打ち込んでいきます。なお、これ以降本巻ではリスト4.1のようにプログラムの行頭に参照用の行番号を付けて掲載しますが、これは実際のプログラムには関係ありませんから入力しないでください。

③ プログラムのセーブとエディタの終了

プログラムをすべて入力し終えたら、それをディスクにセーブしてエディットを終了します。MEDでは、まず **ESC** を押すと画面の最下行に、

```
Command:
```

という表示が現れますから、そこで **q** **c** の順にキーを押してください。すると、さらに、

```
Save (Y/N)?
```

と質問されます。ここで **y** を押すと、プログラムがセーブされ、エディタは終了します。

以上はMEDを使った場合のプログラム作成手順でした。ほかのエディタを利用している方は、それぞれの操作マニュアルをお読みのうえ、プログラムを入力してください。

●プログラムのコンパイル

プログラムを無事に入力し終えたら、CC コマンドによるコンパイル作業に移ります(図 4.2)。ただし、前章のサンプルと違い、今回はみなさん自身の手で打ち込んだプログラムをコンパイルするわけですから、文字のタイプミス等によるエラーの発生がどうしても問題になるでしょう(これだけのサイズのプログラムを一発でノーミスコンパイルできたなら、あなたの今日の運勢は大吉です)。

不運にもコンパイル作業がエラーで中断してしまった場合は、再びエディタを起動して、プログラムの修正を行わなければなりません。次に述べるエラーへの対処法などを参考に、1つずつ誤りをツブしていきましょう。

```

A>cc triangle
      :
      コンパイル過程省略
      :
A>dir triangle
TRIANGLE C      582 89-01-01  1:00p
TRIANGLE COM    6144 89-01-01  1:05p .....コンパイラが作成したオブ
      2 files  486400 bytes free          ジェクトプログラム
A>

```

図 4.2 CC コマンドによる TRIANGLE.C のコンパイル

●エラーへの対処法

コンパイルの途中でプログラムの間違いを見つけると、MSX-C コンパイラはそのエラーの場所と種類を以下の形式のメッセージで知らせてくれます(これは CF コマンドが表示するメッセージですが、ほかのコマンドを実行するときにエラーが発生することはほとんどありません)。

```

line xxx column xxx: .....
      ↑           ↑       ↑
      行の位置   文字の位置 エラーの種類

```

たとえば、「TRIANGLE.C」の7行目を、図4.3のように打ち間違えた場合を考えてみましょう。

```

:
5: {
6:   while (--n >= 0)
7:     putchar(ch); .....正しくはputcharである
8: }
:

```

図4.3 スペリングの間違い

このプログラムをコンパイルすると、MSX-Cは図4.4に示すエラーメッセージを表示します。

```

A>cc triangle
      途中省略
line 7 column 14: undeclared function 'putchr'
errors detected

```

図4.4 エラーメッセージの例

このエラーメッセージは、プログラムリスト7行目の14文字目にエラーが見つかったということを表しています。ただしMSX-Cの出すエラーメッセージでは、行数も文字数も0から数え始めるので注意してください(それに合わせて本書のプログラムには0から始まる行番号を付けました)。

ところで、図4.4のエラーは「未定義の関数 putchar が使用された」という意味ですが、このメッセージからエラーの発生理由がスペリングミスだと見抜くことは、今の段階ではなかなか難しいかもしれません。ですから、もう少しC言語に慣れてプログラムの内容が理解できるようになるまでは、とにかくエラーの行番号を頼りに、リストを1文字ずつ調べて間違いの箇所を見つけるとよいでしょう。

もっとも、CではBASICと違って、エラーメッセージで示された行にエラーの直接の原因があるとは限りません。たとえば図4.5を見てください。

```

:
3: char  ch .....最後にセミコロンが抜けている
4: int   n;
5: {
:

```

図 4.5 セミコロンを打ち忘れた

これは実際には3行目の最後のセミコロンが落ちているのですが、コンパイラがそれをエラーと判定するのは、セミコロン以外のもの、すなわち4行目の `int` が現れた時点です。そのため、このプログラムをコンパイルすると、エラーの場所は4行目だと表示されてしまいます(図 4.6)。

```

A>cc triangle
:
: 途中省略
line 4 column 0: ';' expected ..... 4行目と表示されているが、実際の
errors detected                      エラーの原因は3行目にある

```

図 4.6 エラーメッセージで示された行にエラーがない場合

このように指摘された行にエラーが見つからない場合は、プログラマリストを1行ずつ前にさかのぼってチェックしてください。たぶん近くの行にエラーの原因を発見できるはずです。

またCでは、エラーが1つ発見できてもそこで終わりとせずに、最後までコンパイルを実行し、見つかったすべてのエラーを表示します。そのため、たった1カ所の間違いが次々にエラーを誘発して、エラーメッセージの山が築かれてしまうこともあります。

そのような間違いの例を図 4.7 に示します。これは8行目の閉じブレース記号を閉じカッコとしてしまったものです。


```

5: {
6:   while (--n >= 0)
7:     putchar(ch);
8: ) .....正しくは閉じブレース記号「}」である

```

図 4.7 カッコの対応が正しくない

そして、図 4.7 をコンパイルしたときのエラーメッセージを図 4.8 に示します。本来のエラーは 8 行目の 1 個だけなのですが、ここでボタンが掛け違ったせいで、10 行目の解釈がおかしくなり、その次の行にも影響が及び、結局以降のプログラムはすべて不正なものとなさされてしまいました。

```

A>cc triangle
: 途中省略
line 8 column 0: bad element .....本来のエラー
line 10 column 4: undeclared function 'main' .....これ以降は最初のエラーに
line 10 column 5: undeclared identifier 'argc' 誘発された副次的なエラー
: 途中省略
line 20 column 20: undeclared identifier 'argv'
line 34 column 0: unexpected eof inside function 'putline'

```

図 4.8 副次的なエラーメッセージの山

このように、ちょっと考えられないほど大量のエラーメッセージが出た場合、それらはほとんど、1 個目のエラーが生んだ幻のエラーに違いありません。このようなときは、とにかく最初のエラーメッセージが示す間違いを訂正してみてください。それだけで、たぶんたくさんあったエラーの山は嘘のように消えてしまうでしょう。

●プログラムの実行

無事にコンパイルは終わりましたか？ そうしたら、いよいよ実行です。MSX-C で作ったオブジェクトプログラムは、MSX-DOS の外部コマンドとなり、その名前(ここでは TRIANGLE)を入力するだけで実行できます。

図 4.9 に TRIANGLE コマンドの実行例を示します。この例のとおりに三角形が表示されれば大成功。もしも表示がうまくいかなかった場合は、プログラムのどこかに入力ミスがあるはずですので、再びエディタを起動してリストをよーくチェックしてください。


```
A>triangle   
  +  
  +++  
  +++++  
  ++++++  
  +++++++  
A>
```

図 4.9 TRIANGLE コマンドの実行例

また C では、コマンド名の後ろにパラメータを付けて細かい動作を指定できるようなプログラムが簡単に作れます。このサンプルプログラムでも、適当な数値を与えると三角形の大きさが変わるような工夫をしました。

たとえば図 4.10 の実行例では、コマンド実行時にパラメータ 3 を指定して、3 段の三角形を描かせています。これ以外にもいろいろな数値を指定し、その結果を確かめてください。


```
A>triangle 3   
  +  
  +++  
  +++++  
A>
```

図 4.10 パラメータを指定して TRIANGLE コマンドを実行

■ Cのプログラムの合体パワー

さらに、Cで作られたオブジェクトプログラムには、データの入力元や結果の出力先を変更するI/Oリダイレクションやパイプの機能が、たいへん効果的に利用できるという特徴があります。おかげで、Cでは複数のプログラムを組み合わせると、それぞれを単独で使うときの何倍ものパワーを発揮させられます。

● 第2のプログラムの入力とコンパイル

このことを理解していただくため、もう1つのプログラム「DOUBLE.C」を登場させましょう(図4.11)。さきほどと同様、このプログラムもエディタで入力し、CCでコンパイルしてください。

```

0: /* 1行のデータを2行に出力する */
1:
2: #include <stdio.h>
3:
4: main()
5: {
6:     char line[80];
7:
8:     while (gets(line, 80)) {
9:         line[strlen(line)-1] = '%0';
10:        puts(line);
11:        puts(line);
12:        putchar('%n');
13:    }
14: }

```

図 4.11 サンプルプログラム “DOUBLE.C”

DOUBLE コマンドは、図 4.12 のように、キーボードから1行分のデータを入力するごとに、それを2つ横に並べて表示するという動作を行います。プログラムを終了させるには、行の先頭で **CTRL** + **Z** (**CTRL** キーを押しながら **Z** を押す)を入力します。

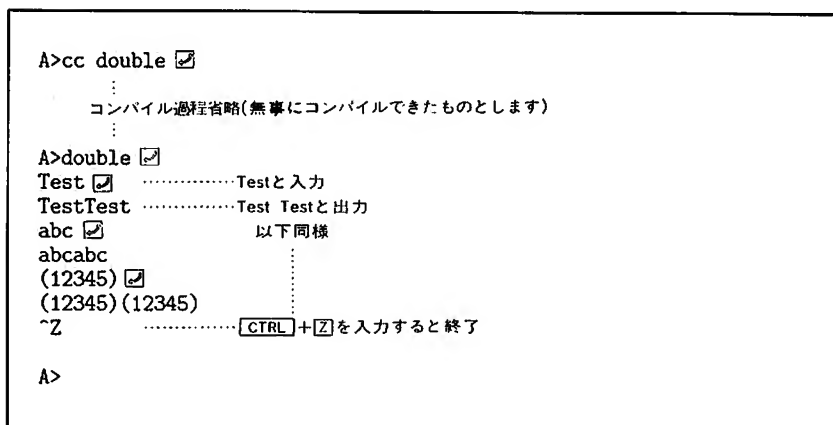


図 4.12 "DOUBLE.C"の実行

●リダイレクションとパイプの利用

さて、これで TRIANGLE と DOUBLE の2つのプログラムが手に入りました。BASIC ならば、プログラムを2つ集めてきても、それはプログラム2個ぶんの働きしかしないでしょう。しかしCのプログラムは、1+1が3にも4にもなる可能性を持っています。

実際に2つの機能を組み合わせるのは、プログラムの入出力リダイレクションとパイプです。これらは各コマンドを実行するときに、表4.1に示す記号を用いて実現されます。

機 能	記 号	意 味
出力リダイレクション (新規書き込み形式)	出力コマンド>ファイル	・指定ファイルにデータを書き込む
出力リダイレクション (追加形式)	出力コマンド>>ファイル	・指定ファイルにデータを追加
入力リダイレクション	入力コマンド<ファイル	・指定ファイルからデータを読み出す
パイプ処理	出力コマンド 入力コマンド	・次のコマンドにデータを受け渡す

表 4.1 リダイレクションとパイプの記号

表 4.1 で「出力コマンド」と表したのは、画面に対して出力を行うプログラムのことで、TRIANGLE も DOUBLE も、どちらもそう呼ばれる資格を持っています。これらに対して出力リダイレクションを行うと、たとえば次のような処理が可能です。

triangle > x	x というファイルに三角形を書き込む
triangle >> x	x というファイルに三角形を追加する
double > y	キーボードから入力したデータを 2 倍して y というファイルに書き込む

「入力コマンド」のほうは、キーボードからの入力を受け取るプログラムのことで、ここでは DOUBLE だけがそれに相当します。

double < x	x というファイルから読み込んだデータを 2 倍して画面に出力する
------------	-----------------------------------

そして出力コマンドと入力コマンドをパイプ記号「|」(SHIFT キーを押しながら ¥ を押す) で連結すると、両者の間でデータの自動的な受け渡しが行われます。

triangle double	三角形を 2 倍して画面に出力する
-------------------	-------------------

●組み合わせの妙を見よ

入力リダイレクション、出力リダイレクション、パイプの三者は、組み合わせで使うこともできます。図 4.13 にいくつか実例を示しました。

```

A>triangle 2 | double > tri .....TRIANGLEの出力をDOUBLEに渡して、
                                     その結果をTRIに書き込む
A>type tri
+ +
+++ +++

A>double < tri | double > tri .....TRIの内容をDOUBLEに読ませ、その
                                     出力結果をまたまたTRIに書き戻す
A>type tri
+ + + + + + + +
+++ +++ +++ +++ +++ +++ +++ +++

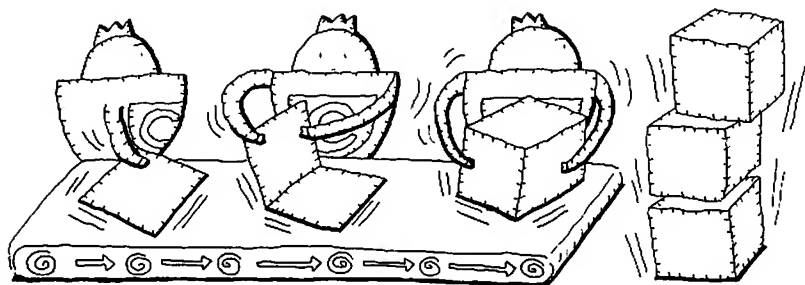
A>triangle 4 | double | double >> tri ...TRIANGLEの出力を2段のDOUBLEに通し、
                                     結果をTRIに追加書き込みする
A>type tri
+ + + + + + + +
+++ +++ +++ +++ +++ +++ +++ +++
+ + + + + + + +
+++ +++ +++ +++
+++++ ++++++ ++++++ ++++++
+++++++ ++++++++ ++++++++ ++++++++

A>

```

図 4.13 パイプとリダイレクションのさまざまな組み合わせ

いかがでしょう、こりやなかなか面白そうだと思うでしょう？ こうしてプログラム同士を有機的に結び付けられる点が、BASICにはなかったCの大きな特徴であり、強みなのです。



42 Cのプログラムの読み方

Cのプログラムがどのように用いられるか、ひとつと説明も済みましたので、そろそろプログラムの中身に話を移しましょう。

ただし、プログラムの個々の命令の意味については、まだ何も説明していませんからそっちに置いとくことにして、ここではごく形而下的なレベルでプログラムを眺めることにします。

■ プログラムの外見を拝見

というわけで、56ページの「TRIANGLE.C」をもういちどご覧ください(エディタを起動して画面にリストを表示させるとよろしい)。このプログラムを見て気がつくことをボンボンとあげてみましょう。

● Cのプログラムは小文字で書く

まず、Cではプログラム全体が小文字のアルファベットで書かれています。BASICでおなじみの条件判断文のキーワード `if` や、ループ文のキーワード `for` も、Cでは必ず小文字で書かないといけません。これを BASIC のように大文字で `IF` や `FOR` と書くとエラーになります。

変数などの名前に大文字を使用することは原則的には自由なのですが、Cのプログラミングの慣習として、やはりこれも小文字のみを使うことになっています。大文字を使うのは、こいつはちょっと普通じゃないのだぞ、ということ強調したいときに限ります。

● Cのプログラムには行番号が存在しない

BASIC では、プログラムを入力するにも `GOTO` 文の飛び先を指定するにも行番号が不可欠でした。ところがCのプログラムからは、ご覧のとおり、

この行番号がすっかり姿を消しています。

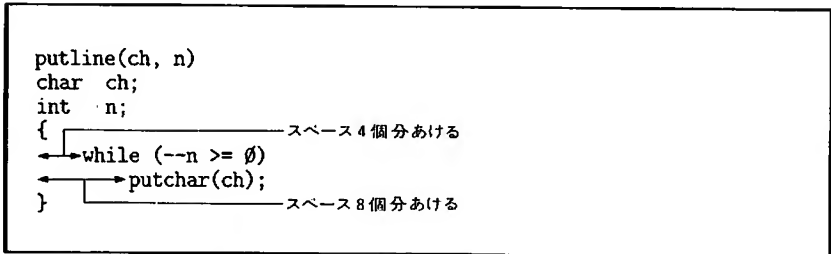
なぜなら、Cではリストの入力にはテキストエディタを使いますし、プログラムの流れの制御にはGOTO文よりもっとわかりやすいwhile文やdo文などが用意され、行番号はまったく必要なくなったからです。

BASICにどっぷり浸かっている方は、行番号がなくなると、なにか自分の立脚点が失われてしまったような、アイデンティティ喪失の危機を感じるかもしれませんが、それは単に気のせいです。慣れればCの自由さのほうがずっと気に入るはずですよ(経験者は語る)。

●空白のバランスは個人の美的センスが決める

ところで、Cには行番号がないところか、そもそも「行」というものに意味がありません。

実はCのプログラムはフリーフォーマットと呼ばれ、どこに空白を入れ、どこで改行をするかが、プログラマの判断に完全にまかされています。たとえば、リストの最初のほうに次のように書かれた部分がありますが……。



```
putline(ch, n)
char ch;
int n;
{
    while (--n >= 0)
        putchar(ch);
}
```

スペース4個分あける

スペース8個分あける

スペース8個分あける

図 4.14 筆者がカッコイイと思うプログラムのフォーマット

あくまでもこれは筆者がかっこよいと思った書き方にすぎず、単語を1字1句この場所に置かねばならないという決まりはないのです。空白や改行がジャマくさいと感ずる方は、次のようなベタ詰め「1行プログラム」を書くことも可能です。

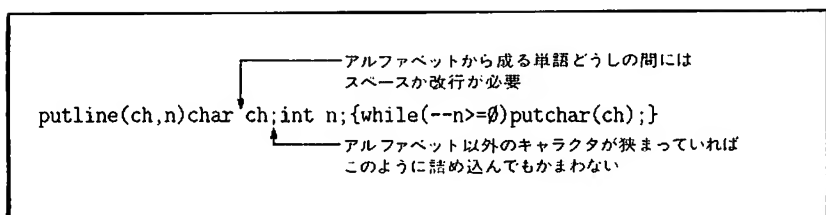


図 4.15 極端な例「ベタ詰め1行プログラム」

唯一の決まりは、「アルファベットから成る単語の間は、少なくとも1個のスペースまたは改行で区切る」ということ。つまり、

```
...charch ; intrn...
```

のように単語を続けて書くのだけはルール違反です(アルファベット以外の記号ならスペースをあけずに続けて書いても大丈夫です)。

行の左端に空白をあけてインデント(字下げ)を付けるときなども、本書では紙面の大きさの関係もあってインデントの深さを基本的に4の倍数と決めています。みなさんがプログラムを入力するときはスペース3個でも5個でもいくつでもかまいません。

まあ要は、自分でわかりやすいと思う形に書くのがいちばんでしょう。いくつかプログラムを作りながら、みなさんも自分なりのプログラミングスタイルというものを見つけていってください。

●実行文はセミコロンで終える

Cのプログラムでは1行にいくらかでも命令を詰め込めると述べてましたが、その反対に、1つの文を複数行に分割することも可能です。たとえば、次のような長い文があるとしましょう。

```
x = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;
```

Cではこれを次のように、文の途中で適当に改行を入れ、見やすく書くことができます。

```
x = 1 + 2 + 3 + 4
    + 5 + 6 + 7 + 8;
```

なぜ、この代入文が「 $x=1+2+3+4$ 」で終わりにならず、「 $x=1+2+3+4+5+6+7+8$ 」だと判断できるかというと、Cには文の最後は必ずセミコロンので終えるという決まりがあるからです。逆にいえば、Cの文はセミコロンが最後に付いていない限り、まだまだ先に続くものとみなされるのです。

ですからCでは、たった1つのセミコロンが、文の意味に大きな影響を与えます。文のどこにセミコロンをつけ、どこには必要ないかということは5章で説明しますが、とりあえずサンプルプログラムを入力するときには、セミコロンの有無にはよく気を付けてください。

関数がプログラムを組み立てる

さて、図4.1をざっと眺めると、このプログラムは大きく分けて2つの部分から成っていることがわかります。すなわち、

```
putline(ch,n)
char ch;
int n;
{
    .....
}
```

という前半のブロックと、

```
main(argc, argv)
int argc;
char *argv []
{
    .....
}
```

という後半のブロックです。厳密に言えば、両者の前にさらに「#include <stdio.h>」と書かれた行がありますが、これはプログラムの最初で必ず唱えるヒラケゴマの呪文のようなものと思って、いまのところは気にしないでください。

この2つのブロックでは、どちらも「関数の定義」を行っています。前者が `putline()` という関数の定義、そして後者が `main()` という関数の定義です（本書では名前のうしろにカッコを付けて関数名を表します）。

●プログラミングの基本は関数の定義である

関数はCのプログラムの基本的な実行単位です。BASICのサブルーチンと似ていますが、名前で呼び出せること、引数をカッコに入れて渡せること、この2点においてCの関数はサブルーチンより何倍も便利です。

たとえば、プログラムの前半で定義している `putline()` は、文字 `ch` を `n` 個並べて表示する関数ですが、もし BASIC で同様な機能のサブルーチンを作り、それを使って文字「A」を10個並べようと思ったら、たぶん次のようなコール命令を書くことになるでしょう。

```
CH$="A": N=10: GOSUB 1000
```

一方、`putline` 関数を使って「A」を10個並べるには、次のように書きます。

```
putline('A', 10);
```

こうすると「A」と10が自動的に `putline()` の使う変数 `ch` と `n` に代入されて、関数が実行されるのです。BASICに比べると、こちらの書き方のほうが圧倒的にスッキリしているでしょう？

さらに、BASICではサブルーチンを行番号で呼び出していたのに、Cでは「`putline`」という機能に沿った名前を使えるため、プログラムを読んだときに一段と理解しやすくなっています。

そして、いったん関数を定義すれば、これはもう新しい命令が出現したのと同じです。実際に、後半の `main()` の中では、この `putline()` が次のように元から MSX-C に用意されている `putchar()` と肩を並べてバシバシ使われています。

```
        :  
        putline(' ', spc) ;  
        putline('+', len) ;  
        putline(' ', spc-1) ;  
        putchar('¥n') ;  
        :
```

いってみれば、Cのプログラミングとは、このような関数を作っては組み立てていくことの大きいなる積み重ねなのです。

● ライブラリ関数はシステム御用達の関数である

こうして自分で定義する関数以外に、MSX-Cのシステムには初めから用意されている基本的な関数がいくつかあり、それらはライブラリ関数と呼ばれます。たとえば「TRIANGLE.C」にも登場した `putchar()` は、文字をひとつ表示するためのライブラリ関数です。

ただし、Cのライブラリ関数というものは、BASICの基本命令がそうであるような特別な存在ではありません。BASICでは、サブルーチンはいくら頑張ってもサブルーチンの身分のままで、PRINT命令やINPUT命令と同じレベルの命令にはなれませんでした。しかしCでは、自作の関数とライブラリ関数に本質的な差はまったくないのです。

実をいいますと、ライブラリ関数とは、かつてどこかの誰か(実際にはCの開発者)が作って「CLIB.REL」というライブラリファイルにストックしておいてくれた関数にすぎません。`putline()`もライブラリファイルに登録すれば立派なライブラリ関数に変身しますし、`putchar()`もライブラリファイルから削除してしまえばもはやライブラリ関数ではなくなります(こういったライブラリ操作は下巻のLIB80コマンドを使うと簡単に実行できます)。

● main 関数からすべては始まる

Cのプログラムでは、これらのライブラリ関数や自作の関数が互いに呼び出したり呼び出されたりして作業を進めていきます。そして、ときにはかなり込み入った関係が作られることもあります。

その例として、図 4.16 に示した「SUMOMO.C」を見てください。

```

0: #include <stdio.h>
1:
2: mo_momo(), momo(), main(), mo(), sumomo();
3:
4: mo_momo()
5: {
6:     mo(); putchar(' '); momo();
7: }
8:
9: momo()
10: {
11:     mo(); mo(); putchar(' ');
12: }
13:
14: main()
15: {
16:     sumomo(); mo_momo();
17:     momo();    mo_momo();
18: }
19:
20: mo()
21: {
22:     putchar('t');
23: }
24:
25: sumomo()
26: {
27:     putchar('s'); momo();
28: }

```

図 4.16 SUMOMO.C

ここでは、次のような5つの関数が定義されています。

mo_momo()	……「モ モモ」と表示
momo()	……「モモ」と表示
main()	……「スモモ モモモ モモ」と表示
mo()	……「モ」と表示
sumomo()	……「スモモ」と表示

だいぶ複雑ですが、どの関数がどの関数を呼び出しているか、わかりますか？ ためしに関数 `mo_momo()` の定義を見てみると…….

```
mo_momo()
{
    mo(); putchar(' '); momo();
}
```

なるほど、この関数は `mo()` と `putchar()` と `momo()` の3つを呼び出しています。そこでさらに `momo()` の定義はどうなっているかというと…….

```
momo()
{
    mo(); mo(); putchar(' ');
}
```

ほほう、ここでもまた `mo()` と `putchar()` を利用しているぞ、などとチェックしていった、すべての関数の呼び出し／呼び出され関係をまとめると、図 4.17 のようなネットワークができあがります。C のプログラムは、多かれ少なかれ、こういった関数呼び出しの連鎖で成り立っています。

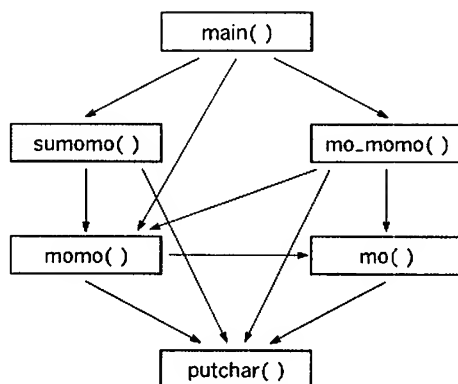


図 4.17 「SUMOMO.C」の関数の相関ネットワーク

さてそれでは、この「SUMOMO.C」をコンパイルして実行すると、どのような結果が得られるでしょう。みなさんも一緒に試してみてください。プログラムに誤りさえなければ、画面には次のメッセージが表示されるはずです。

スモモ モ モモ モモ モ モモ

さあ、そこで疑問が浮かびます。上のように表示されたということは、すなわち `main()` が実行されたわけです。しかし、なぜ `main()` なのでしょう？ 図 4.16 のプログラムでは、合計 5 つの関数を定義しています。その中で、なぜ `momo()` でも `sumomo()` でもなく、`main()` が実行されるのでしょうか？

実は、すべての関数の中で、たったひとつだけ特別な地位を与えられているものが、`main()` なのです。「main」という名前の関数を定義すると、それは自動的にプログラムの実行開始の入口とみなされます。BASIC のプログラムが必ず先頭の行から実行されるものと決められていたように、C では必ず `main()` からプログラムが実行されるのです。これは `main()` の定義がリスト中のどこにあって関係ありません(普通は目立つようにプログラムの先頭か最後に置きます)。

逆にいえば、プログラムの中には必ず `main()` の定義を含める必要があります。そうでないとコンパイラはどこから手をつけていいのか判断できません。そのような(`main()` のない)プログラムを CC でコンパイルすると、L80 によるリンク作業の段階で、`main()` が定義されていない、というエラーメッセージが表示されてしまいます。

■ 用意周到の宣言文

ところで、図 4.16 には、ここまで出てきた他のプログラムと少し違う点があります。すでにお気付きの方もいると思いますが、関数定義の本体が始まる前に、次のような文が書かれているのです。

```
mo_momo(), momo(), main(), mo(), sumomo() ;
```

こりゃなんだ？

●宣言文はコンパイラへの事前連絡である

タネを明かしてしまうと、この文は、これから `mo_momo()`、`momo()`、`main()`、`mo()`、`sumomo()` の5関数が使用されるぞ、という意味の「関数使用宣言」です。

もしこの宣言がなかったらどうでしょう。コンパイラは、コンパイル作業を進めながらプログラムリストを先頭の行から順に見ていきます。そして次のような関数の定義に出会います。

```
mo_momo()
{
    mo(); putchar(' '); momo();
}
```

ところがコンパイラは、突然出てきた `mo()` や `momo()` が何者であるかが理解できません。プログラムを書いた我々自身は、リストのうしろのほうで、これらがちゃんと関数として定義されていることを知っています。しかし、リストを先頭から見てきて今やっとここに到達したばかりのコンパイラは、後方にどんなことが書かれているのか、まったく知見を持ちあわせていないのです。

そこでコンパイラは、この `mo()` や `momo()` を、「私の知らない関数だかなんだかが使用されている」と判断してエラーにしてしまいます。ために「SUMOMO.C」から関数使用宣言の部分の部分を消してコンパイルを行っててください。コンパイラはエラーの山を吐き出してくるでしょう。

この問題に対する解決策は2つあります。1つは関数定義を並べる順番をうまく案配して、どんな関数も必ず「定義が先、使用は後」の方針を守ること。たとえば、本章の最初に紹介した「TRIANGLE.C」では、まずプログラムの前半で `putline()` という関数を定義し、後半でそれを使用していました。これならば、わざわざ関数の使用宣言などしなくとも、コンパイラは文句をいいません。

実をいえば、「SUMOMO.C」でも、関数定義を次の順におこなえば宣言は必要ありませんでした。こうすれば、どの関数も必ず使用される前にその定義が書かれていることになるからです。


```

mo()
↓
momo()
↓
mo_momo(), sumomo() (どちらが先でもよい)
↓
main()

```

しかし、いつでもそう都合よく順番に関数を並べられるとは限りません。たとえば、関数 A が関数 B を呼び出し、関数 B のほうも逆に関数 A を呼び出しているような場合は、どちらの関数定義を先に書いても、結局は一方が未定義のまま使われることになってしまいます。

そこで用いられるのが、もう 1 つの解決策である関数宣言です。これは、「後でこういう関数を使うつもりですので、そこんとこよろしく」と前もってコンパイラに挨拶しておき、万全の準備をさせようという方法です。「SUMOMO.C」の例のように、使用する関数を最初に宣言しておけば、それがどんな順番でプログラム中に現れても、コンパイラは処理してくれるわけです。

なお、関数宣言は「この関数を使うかもしれない」という非常に弱い意味しか持っておりません。ですから宣言した関数を必ずしも実際に使う必要はありませんし、使わなければそれをプログラムする必要もありません。

その実例として、図 4.18 を見てください。

```

#include <stdio.h>

ahaha(), ufufu(); ..... 2つの関数を宣言しているが...

main()
{
    putchar('O'); }
    putchar('k'); } ..... どちらも使われていない
}

```

図 4.18 関数宣言だけしておいて使わない例

ここでは最初に `ahaha()` と `ufufu()` の2関数を使用宣言しています。しかし結局これらの関数は使われもせず、定義もされず、単に名前が出たのみで終わってしまいました。これでもよいのです。関数の使用宣言というのは、イザというときの備えであって、必要なければ無視されるだけです。

●「STDIO.H」はライブラリ関数の宣言をおこなう

さて、関数は定義が出てくる前に無断で使用するとエラーになる、ということを書きました。ここで問題となるのがライブラリ関数です。最前からサンプルプログラムの中で `putchar()` を繰り返し利用しましたが、その使用宣言はまったく行っていませんでした。ライブラリ関数というものは、コンパイラに断わりなく勝手に使ってもよいのでしょうか。

その答えは「No」です。前にもチラッと触れたとおり、Cではライブラリ関数とプログラム自身が定義する関数の間に本質的な違いはなく、ライブラリ関数といえども、やはり無断で使用すれば MSX-C コンパイラは見逃してくれないのです。

ではなぜ、ここまで紹介したサンプルプログラムは無事にコンパイルできたのか。秘密はプログラムの先頭に置かれた次の命令にあります。

```
#include <stdio.h>
```

`#include` は、プログラム中に別のファイルの内容を読み込む命令です。この命令に出会うと、コンパイラは山カッコ<>の中に指定された名前のファイルを読み込み、その内容を現在位置に挿入します(この動作をファイルのインクルードと呼びます)。

ですから、「`#include <stdio.h>`」という命令があると、そこに「STDIO.H」の内容が読み込まれてきます。そして、実はこの「STDIO.H」というファイルの中に、MSX-Cが用意したライブラリ関数の宣言がすべて書かれているのです。

言い換えれば、「`#include <stdio.h>`」という命令は、ライブラリ関数の使用宣言をズラズラ書き並べるのとまったく同じ効果を持つことになります。したがって、この命令さえプログラムの先頭に書いておけば、ここまで見てきたように、ライブラリ関数は自由に利用してかまわないわけです。

● Cでは変数も定義してから使用する

使用に先だって定義あるいは宣言を要するのは関数だけではありません。Cのプログラムでは、変数を使うにも、まずはコンパイラに対してそのことを知らせておく必要があります。

ここまで見てきたサンプルプログラム中にも、すでに変数定義の命令は登場しています。以下のように書かれている部分がみなそうです。

```
int i;
int size, spc, len;
char ch;
char line [80];
```

最初の「int i;」は整数を扱う int 型の変数 i の定義、その次はやはり int 型の変数 size, spc, len を定義し、3 番目では文字を扱う char 型の変数 ch を定義しています。最後の「char line [80];」は少々特殊な形ですが、これは配列変数の定義です(変数の型や配列の扱いについては6章で詳しく説明します)。

これまで見てきたどのサンプルプログラムでも、変数を使う前には必ずこういった定義を行っていることに注意してください。もし定義なしに変数を使おうものなら、即座にエラーと判定されてしまいます。関数の場合もそうでしたが、C コンパイラは素性の知れない存在に対して非常にキビシイのです。BASIC に慣れていると、先生そりゃないよ、変数ぐらい勝手に使わせてよ、と思うのですが、こうした姿勢はバグの発生を未然に防ぐのに大きな効果があることを知ってください。たとえば、

```
int ii;           ←変数 ii の定義
ii = 1234;        ←変数 ii を使用する
```

と書くべきところを、間違えて次のように打ち込んだとしましょう。

```
int ii;
il = 1234;        ←ii を間違えて il と書いてしまった
```

もしこれが BASIC ですと、そうか、新しく「il」という変数を使うんだな

と解釈し(余計なお世話)、プログラムを誤動作させてしまうでしょう。しかしCコンパイラならば、未定義の「il」という変数が使われた、これはエラーに違いないと判断し、そのことをエラーメッセージで教えてくれます。

常に暴走の危険をはらむマシン語なるものを生成するCコンパイラは、こうしてプログラムを致命的なバグから守っているのです。

■ コメントは「/*」と「*/」で囲む

最後にコメント(注釈)の書き方を紹介して、Cのプログラミングスタイルに関する説明を終えることにします。

Cのプログラムでは、次のように「/*」(スラッシュ・アスタ)と「*/」(アスタ・スラッシュ)で囲まれた部分をコメントとみなします。

```
/* コメント */
```

必要ならばコメントは文の途中に入れることもできます。

```
if (energy < 0) /* タイリョクヲ 7カイハクシタ */ gameover=TRUE;
```

↑
厚かましくも文のドまん中に割り込んだコメント

図 4.19 文中にコメントを入れる

また、前に述べたとおり、Cではプログラム中に改行を入れることは自由ですから、次のように複数行に分かれていても、やはり「/*」と「*/」に囲まれた部分はすべてコメントとみなされます。

```
/*
   コレモ ゼ' ソツ' コノソト ..... もちろんコメントも複数行にまたがってかまわない
   ヨロシイデ' スカ(Y/N)?
*/
```

図 4.20 複数行にまたがるコメント

ちょっと凝って次のようなコメントを書いたプログラムを見かけることもあります。この場合は「/*」と「*/」をうまくデザインの一部として取り入れたわけです。

```

/******
 *          *
 * 3j 1g' 7 7/7t * .....重要なコメントは目だたせたい
 *          *
 ******/

```

図 4.21 デザインされたコメント

プログラムをデバッグするとき、必要ない命令を「/*」と「*/」で囲んで一時的にコンパイルを避ける「コメントアウト」の手法もよく用いられます。そのとき注意しなければならないのは、コメントのネスティング、つまり「/*」と「*/」で囲んだ中に、さらにコメントが書かれている次のような場合です。

```

①
↓
/*
monster(n); /* n ^' 77 / モンスター */
*/
↑
④

```

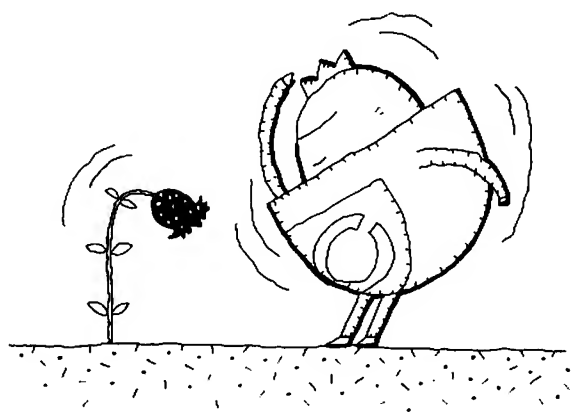
② ③

図 4.22 コメントのネスティング

MSX-Cでは、これは意図したとおりに①から④までがコメントとみなされます。しかしCの処理系によっては、①の次に最初に出て来る「*/」、すなわち③までをコメントとすることもあり、仕様が統一されていません。MSX-C以外のCを使うときには、この点に気を付けてください。

5章

基本的な プログラミング



いよいよ本章から実際のプログラミングの説明にはいります。ただし、これで前章のサンプルに出した「TRIANGLE.C」のようなプログラムが、すぐにも作れるのかなどと期待しないように、この章ではまず、データを入力して処理して出力するだけの、ごく基本的なプログラムの作成から始めましょう。そうして徐々に高度なテクニックへと進んでいくことにします。

前章で紹介したサンプルに匹敵するプログラムを自作するには、皆さんは本書の最後に達するまで、まだまだ何段階ものレベルアップを行う必要があります。プログラミングの道は奥深いのであります。

51

画面にデータを出力する

新しいプログラミング言語に接するとき、最初に覚えるべきものは何かというと、まあこの画面表示ということになるでしょう。なにしろ、コンピュータがどんなにスゴイ計算をしてくれても、その結果がこっちにわからなくては意味がありません。

これからCを覚えようとしている皆さんにとっても、「プログラムをこう書けば結果はこうなる!」と目に見える形で答えが出たほうがプログラムの意味を理解しやすいでしょう。というわけで、プログラミングの第一歩は、画面に文字や数値を表示する方法の紹介から始めます。

■ 1つの文字を表示する —— putchar()

MSX-Cに用意されている最も単純かつ基本的な画面表示の機能は、前章でも登場した `putchar()` による1文字出力です。

ある文字、たとえばaを表示するには、次のように文字の前後をシングルクォート記号で囲み、`'a'`という文字定数の形にして `putchar()` に与えます。

```
#include <stdio.h>
main()
{
    putchar('a');
}

a .....実行結果
```

図 5.1 文字を表示する

この `putchar()` は、BASIC の PRINT 文と異なり、表示後の改行を行いません。そのため、`putchar()` を続けて実行すると、文字は次のように横に並んで表示されます。

```
#include <stdio.h>
main()
{
    putchar('M');
    putchar('S');
    putchar('X');
}

MSX .....実行結果
```

図 5.2 1 文字出力を続けて行う

ここで登場した「文字定数」とは、文字 1 個を表す C 特有の記法です。文字定数のシングルクォートの中には、たった 1 つの文字しか書けません。BASIC には、この文字定数に相当するものはありません。

えっ？ BASIC には文字列というものがあるじゃないかって？ いえいえ、文字列ならば C にもちゃんと用意されています。C の文字定数は文字列とはまったく別の存在なのです（文字定数の驚くべき正体は、本章の後半で明らかになるでしょう）。

■ 文字列を表示する —— `puts()`

さて、理屈の上からは 1 文字出力の機能さえあれば、画面表示に関してはもう何もコワイものはありません。どんな長いテキストを表示するのも、結局は 1 文字ずつの出力の積み重ねです。

とはいえ、ちょっとしたメッセージを表示するたびに、いちいち `putchar()` を書き並べるのも面倒な話ではあります。そこで必要は発明を産み、文字列を表示するためのライブラリ関数、`puts()` が用意されることになりました。

たとえば「C loves You」と表示したければ、次のプログラムのように、この言葉を文字列として puts() に与えます。C の文字列は、BASIC の文字列と同じく、前後をダブルクォートで囲んで表現します。

```
#include <stdio.h>
main()
{
    puts("C loves you");
}

C loves you .....実行結果
```

図 5.3 文字列を表示する

この puts() も前の putchar() と同様に表示後の改行は行いません。ですから続けて実行すると、表示される文字列は次のように横一列につながってしまいます。BASIC の PRINT 文の感覚で使うと失敗するので御用心を。

```
#include <stdio.h>
main()
{
    puts("pot");
    puts("a");
    puts("to");
}

potato .....実行結果
```

図 5.4 文字列表示を続けて行う

■ 数値を表示する —— printf()

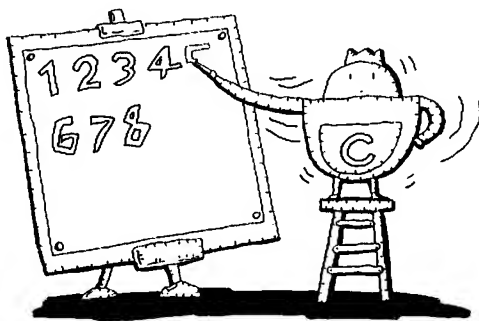
お次は数の表示です。数値の表示には、上の2つとはまた別のライブラリ関数、`printf()`が用意されています。この関数を使って、たとえば1234という数を画面に表示するには次のようなプログラムを書きます。`printf()`も表示後の改行は行いません。

```
#include <stdio.h>
main()
{
    printf("%d", 1234);
}

1234 .....実行結果
```

図 5.5 数値を表示する

さて、ここで不思議なものが現れました。カッコの中の“%d”とはなんぞや？ 1234という数を表示するだけなのに、なぜ妙な記号と一緒に添える必要があるのか？



実は `printf()` はたいへん多くの機能を持った関数なのです。数値を 10 進数として表示するだけでなく、16 進数の形で表示したり、文字コードがその数値に相当する文字 1 個を表示することも可能です。そのため `printf()` を使う場合は、そこで関数にどんな働きをさせたいかという情報も一緒に与えなければなりません。

`“%d”` もその 1 つで、これは「数値を 10 進数形式で表示せよ」という指定に当たります(10 進数以外の表示方法は、これから段々に説明していきますので、しばらくの間お待ちください)。

■ 改行を行う —— ニューライン文字

ここまで述べたように、`putchar()` や `puts()` や `printf()` による表示は、通常は改行を行いません。改行が必要な場合は、`¥n` の記号で表される特別な文字を使用します。

`¥n` は「ニューライン文字」と呼ばれ、C ではこれを画面に出力することによって、改行を行う仕組みになっています。たとえば `puts()` で表示する文字列の最後に `¥n` を追加しておくと、次のように文字列ごとに改行して表示されます。

```
#include <stdio.h>
main()
{
    puts("¥¥¥ ¥n");
    puts("¥/¥n");
    puts("¥ｸ ¥n");
}

¥¥¥
¥/ .....実行結果
¥ｸ
```

図 5.6 ニューライン文字で改行を行う

また newline 文字は、文字列の最後だけではなく、先頭でも真ん中でもどこにでも自由に置くことができます。ですから、1つの puts() で、次のように複数行にわたる出力を行うことが可能です。

```
#include <stdio.h>
main()
{
    puts("イロハニホヘト¥nチリヌルヲ¥n");
}

イロハニホヘト .....実行結果
チリヌルヲ
```

図 5.7 newline 文字を文字列の途中に入れる

この newline 文字は ¥ と n の 2 文字に分けることはできません。これは、あくまでも ¥n と書いて 1 個の文字を表現しているのです。1 個の文字でありますから、シングルクォートで囲んで '¥n' という文字定数を作ることができますし、それを次のように putchar() で出力すれば、やはり改行が行われます。

```
#include <stdio.h>
main()
{
    puts("7カ3タレ7");
    putchar('¥n'); .....putchar( )を使って改行
    puts("7ネナラム");
}

7カ3タレ7 .....実行結果
7ネナラム
```

図 5.8 putchar() で newline 文字を出力する

52 変数とデータの計算法

まとまったプログラムを書こうとすると、どうしても必要になるのが変数というやつです。変数とはナニモノかという点については、すでに BASIC でおなじみでしょうから、「それはデータを格納する入れ物である」などということは、この本ではもうあらためて説明はしませんが、かまいませんね。

■ 文字変数の登場 —— char 型の変数

ということで、さっそく C における変数の使用例を紹介することにします。次に示したリストは、変数 `c` に代入した文字定数 `'X'` を `putchar()` を使って表示するプログラムです。

```
#include <stdio.h>
main()
{
    char  c;

    c = 'X';
    putchar(c);
}

X .....実行結果
```

図 5.9 文字変数を使った文字表示

4 章でも少し触れましたが、C では変数を使用する場合、そのことを前もってコンパイラに対して知らせておかななくてはなりません。上のプログラムでいいますと、次のように書かれた部分が変数 `c` の使用宣言です。

```
char    c;
```

ここで、宣言の最初に書かれている `char` というキーワードは、変数 `c` が文字データを扱うことを指定する型宣言です (`char` は `character` の略)。このことから C では文字データを `char` 型のデータとも呼びます。

なお、BASIC の文字列変数は名前の後ろに \$ 記号を付けて `C$` などと表しましたが、C の変数にはそのような名前による型の区別はありません。char 型の変数も、この次に紹介する `int` 型の変数も、どれもみな同じ外見をしています。慣れないうちは、ちょっと戸惑うことがあるかもしれません。

■ 数値を扱う変数 —— `int` 型の変数

文字データを扱うための変数が `char` 型であるのに対し、数値を扱うための変数は `int` 型と呼ばれます (`int` は `integer` の略)。int 型の変数を使用する場合には、次のような宣言を行います。

```
int     i;
```

次のプログラムは、`int` 型の変数に代入した数値を `printf()` で表示する簡単なサンプルです。

```
#include <stdio.h>
main()
{
    int i;

    i = 1024;
    printf("%d", i);
}
```

1024実行結果

図 5.10 変数を使った数値の表示

C の int 型変数は、BASIC の整数変数と同じく、-32768 から 32767 までの整数を扱います。BASIC の単精度実数や倍精度実数に相当する実数データは MSX-C では扱えません（一般の C では単精度実数にあたる float 型、倍精度実数にあたる double 型が使えるのですが、MSX-C にはそれらの型は用意されていません）。しかし、もともと C は数値計算を行うためのプログラミング言語ではありませんから、さして不便に感ずることはないでしょう。

■ 配列変数の使い方 —— C の配列表現

多くのデータを一括して扱う場合に必要となるのが配列変数です。配列変数も、やはり使用する前に宣言しておく必要があります。たとえば、int 型の要素 3 個からなる配列 a の宣言は、次のように行います。

```
int    a [3];
```

BASIC では、A (3) のように丸カッコで配列を表しましたが、C の配列は、ブラケット記号を使って a [3] と表現するところに注意してください。

次に示すのは配列を使った簡単なサンプルです。このプログラムは、a[0] と a [1] の積を a [2] に代入し、その結果を printf() で画面に表示します。

```
#include <stdio.h>
main()
{
    int  a[3];

    a[0] = 39;
    a[1] = 42;
    a[2] = a[0] * a[1];
    printf("%d", a[2]);
}

15678 .....実行結果
```

図 5.11 配列変数を使う

ところで、このプログラムでは、最初に「int a[3];」という配列宣言を行ったにもかかわらず、実際には a[0] から a[2] までの要素しか使用しませんでした。このことを不審に思ったかたはいませんか？

これは C の配列を使う上で非常に重要な注意点ですから、文字を太くして強調しておきましょう。C の配列では、宣言した大きさより 1 だけ少ない番号の要素までしか使えないのです。

つまり、C の「int a[3];」という配列宣言は、「a[3] までの要素を使う」という意味ではなく、「配列 a は 3 個の要素を使用する」という宣言なのです。そして、C の配列は BASIC と同じく第 0 要素から始まりますから、最後の要素は a[3] ではなく a[2] になります。配列を使うプログラムでは、つねにこの点を忘れないでください。

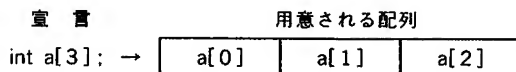


図 5.12 配列宣言と要素数

C と BASIC では、2 次元以上の配列を表現する方法も少し異なります。たとえば BASIC では、2 次元配列を、

A(I, J)

のように表現しましたが、C ではこれを、

a[i][j];

と表します。それぞれの添え字ごとに独立にブラケット記号で囲むわけですが、3×3 の大きさの配列を使用するならば、

int a[3][3];

と宣言することになります。この場合も、a[2][2] までの要素しか利用できません。図 5.13 に 2 次元配列を使ったプログラムの実例を示します。

```
#include <stdio.h>
main()
{
    int a[3][3]; .....3×3の大きさの2次元配列の宣言

    a[0][0] = 0;   a[0][1] = 1;   a[0][2] = 2;
    a[1][0] = 1;   a[1][1] = 2;   a[1][2] = 3;
    a[2][0] = 2;   a[2][1] = 3;   a[2][2] = 4;

    puts("1 + 2 = ");
    printf("%d", a[1][2]);
}
```

図 5.13 2次元配列を使う

さらに、3次元の配列を使いたければ、もうひとつ添え字を追加して、

a [i] [j] [k]

と表現することになります。以下、4次元配列、5次元配列……、もすべて同様です。

■ 加減乗除を行う —— C の算術演算記号

BASICと同様に、Cでも数値データに対して足し算・引き算・かけ算・割り算などの計算を行うことができます。Cに用意された算術演算記号には、次の表に示すものがあります。

演算記号	意 味	BASICでの表現
x + y	x と y を足す	X + Y
x - y	x から y を引く	X - Y
x * y	x と y を掛ける	X * Y
x / y	x を y で割る	X / Y
x % y	x を y で割った余りを求める	X MOD Y
- x	x の正負符号を逆転する	- X

表 5.1 算術演算子一覧

これらはほとんど BASIC の演算記号と一緒にですが、割り算の余りを求める剰余演算子だけは、C 独自の記号を使うので注意してください。これは BASIC では「MOD」と書きますが、C では 1 文字の「%」という記号を用います。

また C では、「-x」と書いて数値の符号を逆転することはできても、「+x」と書いて x 自身を表すことはできません。つまり、

```
i = +123;
```

というような書き方はできないのです。+記号は 2 つの数値を加える用途にしか使えません。

次のプログラムは、加減乗除と剰余演算の簡単な使用例です。BASIC をご存じの皆さんなら、このプログラムの意味はもう説明しなくともわかっていただけるでしょう。

```
#include <stdio.h>
main()
{
    puts("\n 11 + 2 = ");    printf("%d", 11 + 2);
    puts("\n 33 - 4 = ");    printf("%d", 33 - 4);
    puts("\n 55 * 6 = ");    printf("%d", 55 * 6);
    puts("\n 77 / 8 = ");    printf("%d", 77 / 8);
    puts("\n 99 % 10 = ");   printf("%d", 99 % 10);
}
```

```
11 + 2 = 13   実行結果
33 - 4 = 29
55 * 6 = 330
77 / 8 = 9    .....割り算の答は整数に切捨てられる
99 % 10 = 9   .....99÷10の余りは9
```

図 5.14 算術演算子の簡単な使用例

ところで、C の int 型の変数は BASIC の整数型変数と同じく -32768 から 32767 までの範囲の数値を扱うということは前にも述べました。しかし C は BASIC と比べると言語の性格がいくらかイーカゲンにできていて、この範囲を超える数値を int 型の変数に代入してもエラーにはなりません。どうな

るかといえば、誤ったデータを代入したまま、平気で作業をどんどん先に進めてしまうのです。

たとえば次のプログラムを見てください。これは 1000×1000 を計算して答を変数 *i* に代入し、それを `printf()` で画面に表示するだけの簡単なプログラムですが……。

```
#include <stdio.h>
main()
{
    int i;

    i = 1000 * 1000;
    printf("%d", i);
}

16960 .....実行結果
```

図 5.15 代入値が `int` 型変数の範囲を越えてしまう例

この計算の答は 1000000 になるはずなのに、実行してみると結果はごらんのとおり、16960 などというワケのわからない値が表示されてしまいました。1000000 が `int` 型変数の範囲にはいきらなかったのです。答が大きすぎて変数にはいらないならはいらないで、BASIC のように Overflow エラーのメッセージでも出してほしいところですが、C にはそのような親切心は見あたりません。

ですから、C で算術演算を行うときは、その結果が正しい範囲にはいるかどうか、プログラミングに際してつねによく気を付けておかなければなりません。面倒なようですが、このソッケなさが C というものなのです(そう達観できれば C プログラマとして一人前)。

■ ビット操作を行う —— C のビット演算記号

コンピュータがデータを2進数の形で扱っているということは、もうご存じの方も多いでしょう。MSX-Cのint型データも、内部的には次のような16ビットの2進数で表されています。

int型データ	2進内部表現
0	0000000000000000
1	0000000000000001
-1	1111111111111111
12345	0110000001110011
-123	11111111110000101

表 5.2 int 型データの2進内部表現の例

int 型のデータを単純な数値データとして取り扱っている限りは、なにもわざわざ2進数を持ち出してくる必要もないのですが、用途によっては2進表現が役にたつ場合もあります。とくにわれわれ MSX ユーザとしては、グラフィック画面を使ったゲームを作るときに、どうしてもデータをビットパターンとして扱う機能が欲しくなります。

C の演算記号の中には、数値をこのようなビットパターンの形で操作するものもいくつか用意されています。次の表にその一覧をまとめました。

演算記号	意 味	BASICでの表現
$x \& y$	ビットごとのANDをとる	$X \text{ AND } Y$
$x y$	ビットごとのORをとる	$X \text{ OR } Y$
$x \wedge y$	ビットごとのXORをとる	$X \text{ XOR } Y$
$\sim x$	xの全ビットを反転する	$\text{NOT } X$
$x \gg y$	xをyビット右にシフトする	—— なし ——
$x \ll y$	xをyビット左にシフトする	—— なし ——

表 5.3 ビット演算子一覧

この表の中で、「&」「|」「^」「~」の4つは、それぞれ BASIC の「AND」「OR」「XOR」「NOT」とまったく同一の機能を持つビットごとの論理演算子です。これらは数値データの各ビットごとに、次のような演算を行います。

・「&」演算子——ビットごとのAND(2つのビットが両方とも1なら結果は1)

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

例：x の上位 8 ビットをすべて 0 にする

```

x      0011001100110011
y      0000000011111111
-----
x & y  0000000000110011
  
```

・「|」演算子——ビットごとのOR(2つのビットのうち一方でも1なら結果は1)

x	y	x y
0	0	0
0	1	1
1	0	1
1	1	1

例：x の上位 8 ビットをすべて 1 にする

```

x      0011001100110011
y      1111111100000000
-----
x | y  1111111100110011
  
```

・「^」演算子——ビットごとのXOR(2つのビットが異なっているとき結果は1)

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

例：x の下位 8 ビットだけを反転する

```

x      0011001100110011
y      0000000011111111
-----
x ^ y  0011001100110011
  
```

・「~」演算子——ビットごとの反転(元のビットが0のとき結果は1)

x	~x
0	1
1	0

例：x の全ビットを反転する

```

x      0011001100110011
-----
~x     1100110011001100
  
```

図 5.16 ビットごとの論理演算

また表の最後に示した2つの演算子「>>」と「<<」は、ビットシフト演算子と呼ばれ、データのビットパターンを指定した数だけ右または左にずらす働きがあります(同様な演算子は BASIC には存在しません)。この動作を図で表すと次のようになります。

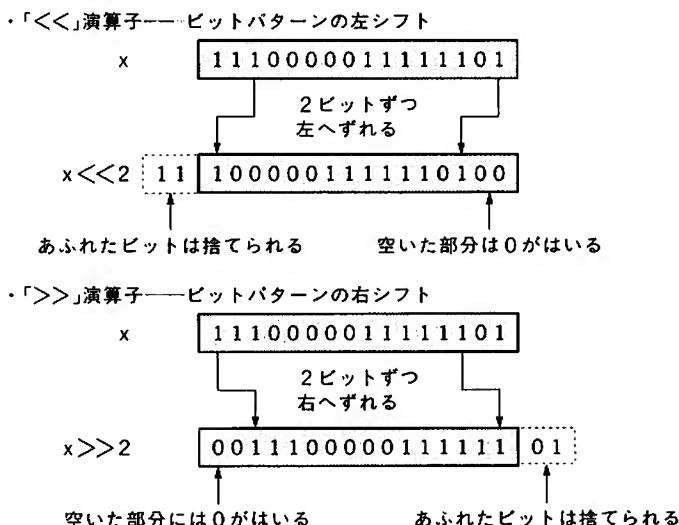


図 5.17 ビットシフト演算

■ 文字データに手を加える —— char 型変数の操作

int 型のデータの操作方法はひとつおりの説明も終わりました。そこで今度は、char 型データに対する操作について述べることにしましょう。といっても実は、char 型データのためにとくに用意された演算というものは、C には1つもありません。

BASIC の場合は、文字列データと数値データはまったく別の存在でしたから、文字と数値の混合計算は許されませんでした。次のような BASIC プログラムを実行するとエラーで中断してしまうことは、皆さんご存じのとおりです。


```

100 C$ = "X"
110 C$ = C$ + 1      ← 「タイプミスマッチ」エラーが発生
120 PRINT C$

```

ところが C の char 型データは、int 型のデータと一緒に計算してもエラーにはなりません。次のプログラムでは、まず変数 c に 'X' を代入し、その後「c = c + 1;」という代入文で c に 1 が加えられ、最後にそれを putchar() で出力しています。この結果として、画面には Y の文字が表示されるはずです。

```

#include <stdio.h>
main()
{
    char c;

    c = 'X';
    c = c + 1;
    putchar(c);
}

Y

```

図 5.18 char 型データと int 型データで演算を行う

足し算だけではありません。四則演算でもビット演算でも、int 型の数値データに対して行えることは、実は char 型データに対してもまったく同様に実行できるのです。

これは、C コンパイラが char 型のデータを、実際には数値として扱っているからです。たとえば文字定数 'X' は、コンパイラにとっては、X の文字コードに相当する 88 (16 進数では 58h) という数値データでしかありません。

この 'X' という存在を、文字 X として見るか、88 という数値として見るかは、時と場合とプログラマの判断によります。たとえば、

```

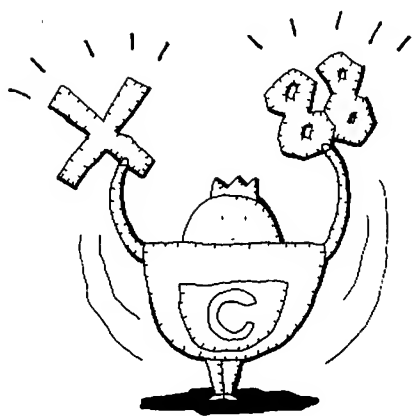
c = 'X';
c = c + 1;

```

を実行するときは、cを数値と考えているわけですし、それを、

```
putchar(c);
```

と表示するときは、今度は変数cの内容を文字と考えているのだといってよいでしょう。Cによるプログラミングは、このあたりも結構イーカゲンです(それがCのよいところでもあるのですけどね)。



53 キーボードから入力する

コンピュータの仕事は、せんじ詰めればどれも入力・演算・出力の3要素で成り立っているものです。これは銀行のキャッシュカード端末(金額指定・勘定・支払い)から、家庭用ゲームマシン(スティックを倒す・敵機はいないか・いざ進め)まで、コンピュータと名が付くものならば変わることはない真実です。

本章では、画面出力、データの計算と、ここまでコンピュータの処理の流れを出口のほうから逆にたどってきました。これで最後に入力の方法さえ覚えてしまえば、もう完璧だ！

■ 押されたキーをすぐに読む —— `getch()`

MSX-Cには1文字入力用のライブラリ関数がいくつか用意されていますが、いちばん使い方がわかりやすいのは `getch()` だと思います。次のプログラムに `getch()` の使用例を示しました。

```
#include <stdio.h>
main()
{
    char c;

    c = getch();
    puts("¥n?キ? キーハ ["");
    putchar(c);
    puts("] ?'ス?");
}
```

図 5.19 1文字入力を行う

このプログラムを実行すると、カーソルを表示して入力待ちになりますから、適当にキーを選んで押してみてください。ここで、たとえば a のキーを押したとすると、すぐに、

```
アナタガ オシタ キー ハ [a] デスネ
```

と表示してくるでしょう。getch()によって1文字入力が行われたのです。なお、getch()自身は押したキーを画面にエコーバックしないことに注意してください。

getch()は、ここまで登場した他のライブラリ関数と異なり、戻り値に意味がある関数です。関数の戻り値とは、たとえば、

```
c = getch();
```

のように書くとき変数cに代入される値です。

BASICでは、INT関数などのように戻り値を返すものを「関数」、PRINT命令のように単に実行されるだけで値を返さないものを「ステートメント」と分けていましたが、Cにはその区別はありません。基本的にCの関数は、どれも戻り値を返します。ただ、その戻り値が意味を持つかどうかの違いがあるだけです。

たとえば、これまで使ってきた putchar() や puts() も戻り値がないわけではありません。これらの関数は、文字の出力に成功すれば0を返し、失敗すると(出力をファイルに向けてリダイレクションしたがディスクが満杯であった場合など)0以外の値を返してそのことを知らせます。ですから、

```
error = putchar();
```

などという代入文を書くことも実はできるのです。ただし、画面にデータを表示する場合は出力に失敗するということがないため、ここまで見てきたように putchar() や puts() の戻り値は無視してしまってもかまいません。

ちょっと話題がずれてしまいました。話を getch() に戻しましょう。この関数は、キーボードが押されるのを待って、そのキーに相当する文字を返します。BASICのINPUT\$関数をご存じの方は、それとちょうど同じ動作をすると思っていたければ間違いありません。

なお、MSX-C の Ver.1.1 では、`getch()` や `getche()` は完全にキーボード入力専用の関数となっているため、この関数を使ったプログラムは、実行時に入力リダイレクションを行ってファイルからデータを読み込むことはできません。MSX-C Ver.1.2 からはこの制約はなくなり、`getch()` や `getche()` への入力もリダイレクション可能となりました。

■ 入力と出力のカラミ合い —— バッファリングの問題

MSX-C Ver.1.2 を使用している方は、`getch()` による入力を行う場合に気を付けなければいけないことがあります。これは実際には Ver.1.2 だけに対する注意点なのですが、C でプログラミングしていればかならずいつかは出会う問題ですから、Ver.1.1 のユーザーの方もぜひ知っておいていただきたいと思います。

この問題とは、画面への文字表示が、文字表示関数の実行と同時にには行われないことです。たとえば、次のプログラムを見てください。

```
#include <stdio.h>
main()
{
    char c;

    puts("INPUT>");
    c = getch();

    puts("%n[");
    putchar(c);
    puts("[ ] が'オサレマシタ'");
}
```

図 5.22 Ver.1.1 と Ver.1.2 では動作が異なるプログラム

ざっと眺めたかぎりでは、このプログラムは以下のような動作を行うものに思えます(事実、MSX-C Ver.1.1 では、プログラムの動作はここに述べたとおりになります)。

- ① 「INPUT>」というプロンプトを表示し入力待ちになる
- ② そこでaのキーを押してみる
- ③ 「[a] ガ オサレマシタ」と表示する

ところが、上のプログラムをMSX-C Ver.1.2でコンパイルして実行してみると、期待通りの結果は得られません。つまり、

- ① 何も表示されずに入力待ちになる
- ② しょうがないのでaのキーを押してみる
- ③ 「INPUT>」と「[a] ガ オサレマシタ」の2行が一度に表示される

という動作になってしまうのです。

この理由は、MSX-CのVer.1.2では、画面出力がバッファリングされているからです。バッファリングとは、putchar()などの関数で出力されたデータをすぐに画面に表示せず、バッファと呼ばれるメモリ領域の中に蓄えておき、バッファがいっぱいになるか、あるいは改行が行われたところで、それをいっきに画面に表示するという方法です。

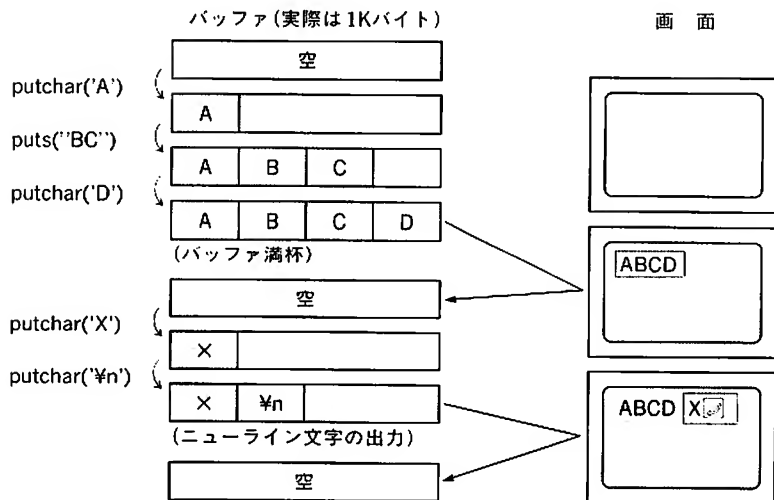


図 5.23 バッファリング出力の動作

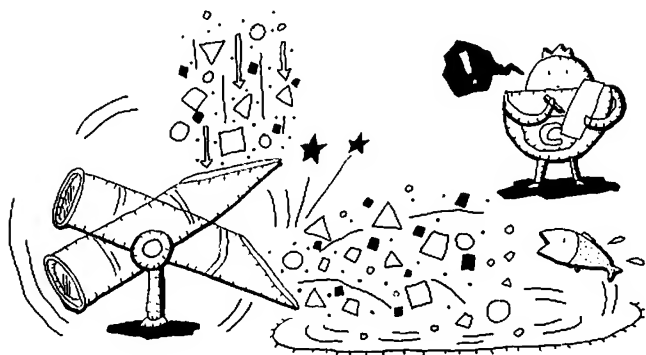
一般に、バッファリングを行うと、1文字ずつ表示を行うよりも素早い動作が期待できます。しかし上のプログラムで見たとおり、画面表示がバッファリングされていると、あるデータを今すぐ表示して欲しいというときに、それが表示されないという不都合も起こります。これはとくに、画面表示とキーボードからの入力交互に進行するようなプログラムで問題になるでしょう。

MSX-C Ver.1.2では、このような不都合を避けるために2つの方法が用意されました。1つは画面出力のバッファリングを完全にやめてしまう方法、もう1つは必要な時点で強制的にバッファの内容を画面に吐き出させる方法です。

まずバッファリング機能を停止するには、`setbuf()`というライブラリ関数を使用します(この関数はVer.1.1には存在しません)。プログラムの最初に1回だけ、`setbuf()`を次のような形で実行しておくと、画面出力はバッファリングを行わないモードに設定されます。

```
setbuf(stdout, NULL);
```

ここで、`stdout`は「標準出力」、`NULL`は「ヌルポインタ」を表す単語ですが、これらの意味についてはファイル操作やポインタの説明を行ってから再び詳しく説明します。いまのところは、上のように書けば画面出力のバッファリングが取りやめになるとだけ覚えておいてください。



それでは、この `setbuf()` を使って、さきほどのプログラムを書き直してみましょう。

```
#include <stdio.h>
main()
{
    char c;

    setbuf(stdout, NULL); .....Ver.1.2ではこの1行を追加する

    puts("INPUT>");
    c = getch();

    puts("%n[");
    putchar(c);
    puts("] が オサレマシタ");
}
```

図 5.24 Ver.1.2 で画面出力をバッファリングさせない

こうして `setbuf()` を 1 行追加するだけで、`puts()` の出力は関数を実行した瞬間に画面に表示されるようになりますから、前のプログラムのようにプロンプトメッセージがバッファに溜ってしまうことはなくなります。

バッファリングの悪影響を避けるもうひとつの手段は、`fflush()` を使う方法です(この関数も Ver.1.1 には存在しません)。`fflush()` は、その時点でバッファに溜っているすべてのデータを強制的に出力させる働きがあります。この動作は「バッファをフラッシュする」とも呼ばれます。

`putchar()` や `puts()` の出力データを画面に表示させるには、`fflush()` を次の形で使用します。

```
fflush(stdout);
```

前の `setbuf()` は、プログラムの最初に 1 回実行しておけばよかったのですが、この `fflush()` は、次のプログラムに示すように、画面表示が必要となるごと(要するに入力の直前ごと)に実行しなくてはなりません。

```

#include <stdio.h>
main()
{
    char  c1, c2;

    puts("INPUT first");
    fflush(stdout); ..... puts( )の出力をフラッシュする
    c1 = getche();

    puts("\nINPUT second");
    fflush(stdout); ..... puts( )の出力をフラッシュする
    c2 = getche();

    puts("\n[");
    putchar(c1);
    puts("] ↑ [");
    putchar(c2);
    puts("] が'オレマシタ'");
}

```

図 5.25 バッファをフラッシュする

なお、プログラムが終了するときには自動的にすべてのバッファがフラッシュされますから、プログラムの最後に行う画面表示については、`fflush()`は必要ありません。

■ 文字列を入力する —— `gets()`

1行ぶんの文字列をまとめて入力したい場合には、`gets()`というライブラリ関数が用意されています。前に紹介した`getch()`は、BASICでいうとINPUT\$関数と同じ機能を持っていたわけですが、こちらの`gets()`はLINEINPUT命令に相当する働きをします。

図 5.26 に、`gets()`を使った文字列入力の簡単なサンプルを示しました。

このプログラムを実行するとカーソルを表示して入力待ちの状態になりますから、適当な文字列をキーボードから打ち込んで、最後にリターンキーを押してください。

```
#include <stdio.h>
main()
{
    char s[200];

    gets(s, 200);
    puts("ｲﾝﾌﾟｯﾄ ﾀﾞﾚﾀ ﾓｼﾚｯﾄ ﾊ --- ");
    puts(s);
}
```

図 5.26 文字列入力を行う

たとえば「abc」を入力すると、プログラムは、

インプット サレタ モジレツ ハ --- abc

と応答を返してくるでしょう。

文字列を入力するときに問題となるのは、CにはBASICのような文字列変数が用意されていないことです。Cのchar型の変数は、BASICの文字列変数と異なり、たった1個の文字しか入れておけません。

ではどうしたらよいのか？ その答は配列の利用です。Cではchar型の配列変数を使って、文字列を格納したり文字列の操作を行う仕組みになっています。たとえば、上のプログラムの中で、

```
char s[200];
```

と書かれた部分は、200文字ぶんの大きさを持つchar型の配列変数sの宣言です。そして、

```
gets(s, 200);
```

というgets()の関数呼び出しによって、キーボードから入力した1行ぶんのデータが配列sに代入されます。なお、200という数値は文字列の長さの最大値を指定するもので、この文字数を超えた入力データは切り捨てられます。

こうして配列変数sに格納された入力データは、puts()を使って再び画面に表示することもできます。それを行っているのが、プログラムの最後の、

```
puts(s);
```

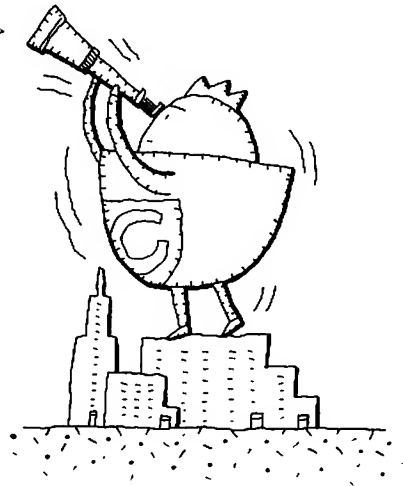
という関数呼び出しです。

ところで、このプログラムに示したように、`gets()`や`puts()`に配列 `s` をパラメータとして与える場合、`s[0]` あるいは `s[200]` などとはせず、ブラケット記号を取り去った裸の `s` という配列名だけを渡しています。これは `s` という配列全体を指定しているのです。

BASIC の配列は、`S(0)` や `S(200)` という個々の要素として利用はできませんでしたが、全体を一括して取り扱うことはできませんでした。それに対して C では、上の例のように 1 個の配列をまるごと操作の対象として扱えるようになっていました。この機能を利用すると、いろいろと面白いプログラミングテクニックも考えられるのですが、その実例は章をあらためて紹介することにしましょう。

6章

Cのプログラムは 如何にして 組み立てられるのか



プログラムの大事な要素の1つに、処理の流れを変える機能があります。ある決まった動作を上から順に実行するだけでは、どんなプログラムもたいしたことはできません。状況に応じていろいろと対応を変えられなくてはプログラムなんて意味がない！ とさえいえるかもしれません。

ところで一口に「処理の流れ」といっても、細かく見れば、世の中に存在するプログラムの数と同じだけの処理の流れがあるわけですが、それらは結局のところ、条件によって動作を選択する「条件判断」か、ある動作を繰り返す「繰り返し」の2種類に大きく分けられます。

以下この章では、その2つの処理をCのプログラムで実現する方法について説明していきます。

61 条件判断

ファジー(あいまい)論理とやらがトレンドな昨今ですが、白か黒か2つに1つの答を選択する条件判断は、やはりプログラミングには欠かせない要素です。イエスかノーかの単純な選択といってもバカにはできません。イエスノーの質問も、20回繰り返せば森羅万象が弁別できるのです(しかし「二十の扉」などという太古の番組は、今や誰も知らないか)。

■ プログラムの最小要素 —— 文

条件判断の説明にはいる前に、プログラムの流れの最小単位である「文」について、きちんと押さえておきましょう。文、などと改まって書くとどうも小難しくなっていけませんが、要するに「ある処理を実行するもの」をCではすべて文と呼んでいます。

ここまでの説明で出てきた文には、次の2種類がありました。

・代入文 (変数にデータを代入する)

```
i = 1234 ;  
c = 'a' ;  
c = getch() ;
```

・関数呼び出し文 (関数を単独で呼び出し、処理を行う)

```
putchar('a') ;  
puts("abc") ;
```

これらの文は、最後がセミコロン「;」で終わっているのが大きな特徴です。セミコロンは、代入文や関数呼び出し文の一部であって、これを含めて文が成り立っているものと考えてください。

さて、こうして今まで出てきた文を2種類にまとめてみると、実はこれでCのプログラムの本質はすべて尽くされてしまいました。Cのプログラムとは、要はこの2つの単純な文を巧妙に組み立てたものにすぎません。その組み立てを行うものが、これから述べる「条件判断」と「繰り返し」なのです。

■ 条件の判定 —— if 文

Cの条件判断の基本となるのはif文です。Cのif文の書式は次のとおりです。

if (条件) 文

このif文の意味は、「条件」が成立すれば「文」を実行せよ、ということです。BASICのIF文とよく似ていますが、Cのif文では条件の後ろにTHENに相当するキーワードを書く必要はありません(書いてはいけません)。また条件はかならずカッコに入れて書かなくてはなりません。

図6.1に示すのはif文を使ったプログラムの例です。このif文は、s以上のキーが押されれば、「ピンポーン」と表示します。

```
#include <stdio.h>
main()
{
    char c;

    c = getche();

    if (c >= 's')
        puts("\n\t' s' ->");
}
```

図6.1 if文による条件判断

このプログラムに示したように、Cのif文はBASICのIF文と違い全体を1行に詰め込む必要はありません。Cのプログラムはフリーフォーマット形式と呼ばれ、プログラム中の単語の並べ方が、すべてプログラムを書く人間

のセンスにまかされていることは前にも述べたとおりです。

ただし一般には、次のような書き方がよく用いられます。本書でもこの形式を採用することにしました。

```
if (c >= 's')      ← if から条件までを 1 行に書く
    puts("¥n ビンボーン");
    ↑
    実行文は 1 段(本書では 4 文字)下げて次行に書く
```

さて、条件が成立しなかった場合にもなんらかの動作をさせたい場合には、else 付きの if 文を使用します。else 付きの if 文の書式は次のとおりです。

```
if (条件) 文 1  else 文 2
```

この else 付きの if 文は、「条件」が成立したときに「文 1」、成立しなかったときは「文 2」を実行します。else に釣られて BASIC の IF 文のクセが出て、くれぐれも if の後ろに then などと書かないように(コンパイラに怒られます)。

図 6.2 は else 付きの if 文の実例です。else 付きの if 文は、このように if と else の頭をそろえて書くとプログラムが見やすくなります。

```
#include <stdio.h>
main()
{
    char  c;

    c = getche();

    if (c >= 's')
        puts("¥nt' /#' -/");
    else
        puts("¥n? ---?");
}
```

図 6.2 else 付きの if 文による条件判断

if文は、それ自体で1つの文とみなされます。ですから、if文の実行文として、さらに別のif文を書くこともできます(このような状態をif文の「入れ子」と呼びます)。

たとえば図6.3のプログラムを見てください。これは押されたキーが数字の1から5までの範囲ならば「ヨシ」と表示し、そうでなければなにもしないプログラムですが、いま述べたif文の入れ子を利用しています。

```
#include <stdio.h>
main()
{
    char c;

    c = getche();

    if (c >= '1')
        if (c <= '5')
            puts("ヨシ");
}
```

図6.3 if文を組み合わせる「入れ子」

また、こういったif文の入れ子構造の中にelseがはいる場合、Cではそのelseはかならず直前の(ただしelseを持っていない)ifにつながるものと決められています。たとえば図6.4のようなプログラムでは、elseは2番目のifに対応するものとみなされます。

```
if (c >= '3') ←-----
    if (c <= '6') ←-----
        puts("¥n3 ト 6 / 7 イダ' テ' ス");
    else ←-----
        puts("¥n7 イダ' ョウ' テ' ス");
```

このifとelseが
対応する

こちらは無関係

図6.4 if文の入れ子とelseの対応

■ 文をまとめる —— 複文

if 文の条件が成立したとき、あるいはしなかったときに、複数の動作を行わせなければ、それらの実行文を「ブロック」にまとめます。ブロックとは、実行文を次のようにブレース記号「{」と「}」で囲んだものです。また、このように複数の文をブロックにまとめたものを「複文」とも呼びます。なおブロックの後ろにはセミコロンを付ける必要がありません。

```
{ 文1 文2 文3 …… }
```

↑ここにはセミコロンは必要ない

次のプログラムに複文の使用例を示します、ここでは if 文の条件が成立したとき、複文を使って 3 つの関数呼び出しを行わせています。

```
#include <stdio.h>
main()
{
    char    c;

    setbuf(stdout, NULL); .....MSX-C Ver.1.2にのみ必要
    puts("[s] い'ヨリノキーヲオシテクダサイ : ");
    c = getche();

    if (c >= 's')
        { puts("¥nj-イス! "); putchar(c); puts(" ^ s い'ヨリテ'ス!"); }
}
```

3つの関数呼び出しをまとめた複文

図 6.5 複文の使用例

ところで上のプログラムでは、複文というものが 1 個の文であることを示すため、全体をわざと 1 行に並べて書きました。しかしこのような書き方はあまり見やすいものではありません(でしょ?)。そこでエレガントなプログラミングを志すわれわれとしては、これをなんとか見やすく美しいプログラムにしようと努力するわけです。

複文の見やすい書き方には多くの流派があり、プログラマが 10 人いれば 10 通りの書き方があるといわれるくらい、その表記法はさまざまですが、本書では原則として図 6.6 の書式を採用することにしました。

```
if (c >= 's') {  
    puts("¥nf-1s! ");  
    putchar(c);  
    puts(" ^ s 1' 3' 1's");  
}
```

図 6.6 本書で採用する if 文の書式

また、この後ろにさらに else が続く場合には、図 6.7 のような位置に else を置きます。

```
if (c >= 's') {  
    puts("¥nf-1s! ");  
    putchar(c);  
    puts(" ^ s 1' 3' 1's");  
}  
else { .....本書ではこの位置にelseを書く  
    puts("¥nf' 1' 3' ");  
    putchar(c);  
    puts(" ^ s 3' 1' 1' ");  
}
```

図 6.7 本書で採用する else 文の書式

■ データの大小を比較する —— 比較演算子

前項のプログラムには、データの大小を比較する「>=」という比較演算記号が登場しました。これは BASIC の場合と同様に「大きいか等しい」という意味を持っています。

C と BASIC の記号の使い方がこの調子でまったく同じならば、皆さんも筆者も幸福だったのですが、実際には C で使う比較演算記号は BASIC のそれと少々異なる部分があります。そこで、条件判断の説明を先に進める前に、C の比較演算記号をまとめて紹介しておくことにしました。表 6.1 が C の比較演算記号の一覧です。

比較演算	意 味	BASICでの表現
$x > y$	x は y より大きい	$X > Y$
$x \geq y$	x は y より大きい等しい	$X \geq Y$ (または $X = > Y$)
$x < y$	x は y より小さい	$X < Y$
$x \leq y$	x は y より小さい等しい	$X \leq Y$ (または $X = < Y$)
$x == y$	x は y と等しい	$X = Y$
$x != y$	x は y と等しくない	$X < > Y$ (または $X > < Y$)

表 6.1 C の比較演算記号一覧

この中でとくに注意が必要なのは、2 つの値が等しいことを表す「==」という記号です。

BASIC では、X の値が Y に等しいことを表すにも「X=Y」、X に Y を代入する場合も「X=Y」と書きましたが、C では両者をきちんと区別しなくてはなりません。上の表に示したように、C のプログラムでは、x と y が等しいということは、

$x == y$ (x と y は等しい)

とイコール記号を 2 つ重ねて書きます。

そして x に y を代入するときだけ、

x = y (x に y を代入)

と BASIC と同じ代入記号(イコール記号1個)を使うのです。

この2つの記号の使い分けは非常に重要です。というのは、Cでは比較記号「==」と間違えて代入記号「=」を使ってもエラーにはならず、それなりの処理が実行されてしまうからです。たとえば図6.8のプログラムを見て下さい。

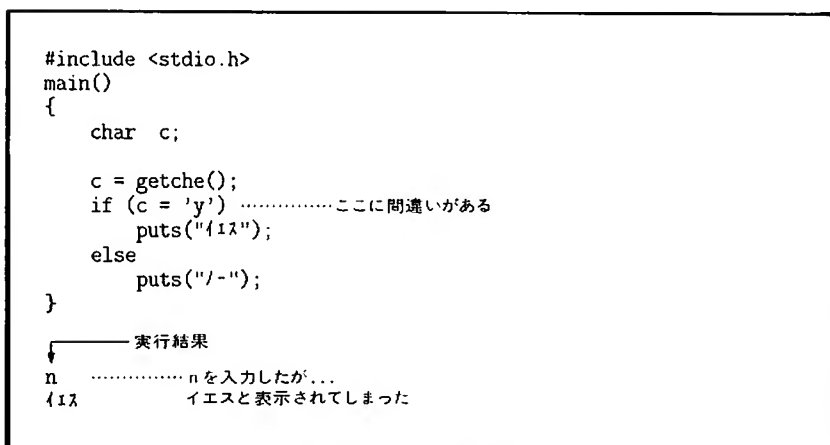


図 6.8 比較演算と代入を間違えたプログラム

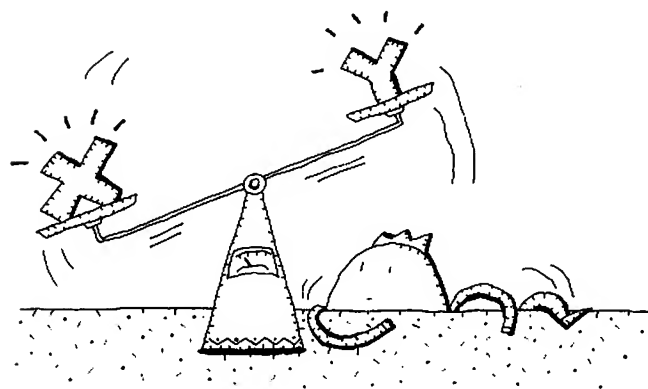
これは、キーボードから y が入力されればイエス、それ以外の文字の場合はノーと表示するプログラムを作ったつもりなのですが、実行してみると、どのキーを押してもイエスの答しか返ってきません。

その原因は if 文の中の「==」と「=」の書き間違いです。つまりイコール記号が1個足りないばかりに、「変数 c が'y'に等しければ」という条件ではなく、「変数 c に代入した'y'という値が真ならば」という意味に化けてしまったのです(後で述べるように C の条件判断では 0 以外の値はすべて真とみなされますから、この場合 if 文の条件は常に成立してしまうことになります)。

慣れないうちはどうしても BASIC のクセが出て、「==」を使用すべき

場所で、つい代入演算子「=」を使ってしまうことも多いと思いますが、くれぐれも注意を怠らないようにしてください。

また、Cでは x と y が等しくないことを「 $x!=y$ 」と表します(!と=を重ねれば \neq になる、と考えると覚えやすいでしょう)。大小比較の記号についてはBASICと一緒に、ただし、BASICでは「より大きいか等しい」ということを表すのに「 $=>$ 」という記号が使えましたが、Cには「 $>=$ 」のほうしか用意されていません。同様に「より小さいか等しい」を表す記号も「 $<=$ 」という書き方しか許されません。



6 2

複雑な条件判断

C の if 文は、初めのうちは BASIC の IF 文と比べると、どうも難しく感じられるようです。とくに人の作ったプログラムなどを見ると、if 文の条件に尋常ではない式が書かれていて思わず頭をかかえることが多い。そんな場合は大抵、ここで紹介する論理値、論理演算子、代入式のどれかがカラんでいるはずです。というわけで、ここではその 3 つのアイテムの説明を行います。

■ 条件判断のしくみ —— 論理値

ここまでは、if 文の条件として、いわゆる「条件式」だけを使ってきましたが、実は if 文のカッコの中にはどんな計算式を書いてもかまいません。たとえば次のような if 文も、立派な C のプログラムなのです。

```
if (error)
    puts("ガチョーン");
```

このような場合、条件が成り立つかどうかは、カッコ中の式の値によって以下のとおりに判定されます。

式の計算結果が 0 以外 —— 条件は成立
式の計算結果が 0 —— 条件は不成立

ですから上の if 文は、変数 error の値が 0 でなければ「ガチョーン」と表示するという意味になります(0 以外ならば、1 でも 2 でも -1 でも結果に変わりありません)。

このように、ある数値データが 0 かそれ以外か、つまり if 文の条件を成立させるかどうかという点だけに注目することを、「データを論理値として扱う」といいます。そしてデータを論理値として扱うとき、0 以外の値を「真」、

0を「偽」と呼びます(表 6.2)。

論理値	その実態
真	0 以外の数値
偽	0

表 6.2 論理値とその値

C の if 文は、原則としてこの論理値を仲立ちとして実行されるしくみになっています。これは通常の条件演算の場合も例外ではありません。たとえば、

```
if (c <= 'l')
    puts("チイサスギマス");
```

という if 文は、まず c と 'l' を比較して真または偽の値を得て、それを if 文が判断する 2 段階の処理になっているわけです。

なお、比較演算の結果としての「真」の値は 1 と決っています。つまり比較演算の答は、かならず 0 または 1 になるのです。これは次のプログラムで確かめることができます。

```
printf("%d", (2 > 1));    ← 真なので 1 が表示される
printf("%d", (2 < 1));    ← 偽なので 0 が表示される
```

■ 条件を組み合わせる —— 論理演算子

複雑な条件判断を行うには、if と else を組み合わせて使うのもひとつの方法ですが、もうひとつ、「A かつ B」や「A または B」などという、いわゆる論理演算を利用するという手もあります。

BASIC の場合は論理演算にはビット演算と同じ記号(AND, OR など)を使いました。それに対して C では、すでに紹介した「&」や「|」などのビット演算子とは別に、論理演算専用の演算子が用意されています。その一覧を表 6.3 に示します。

演算記号	意 味
<code>x & y</code>	x が真で、かつ y も真(論理AND)
<code>x y</code>	x が真か、または y が真(論理OR)
<code>! x</code>	x ではない(論理NOT)

注：「論理XOR」に相当する演算子はありません。

表 6.3 論理演算子一覧

● 「&&」演算子 —— それに加えて～なら(論理 AND)

「`x && y`」は、`x` という条件が成立し、かつ `y` という条件も成立することを表します(「&」と「&」の間には空白を入れてはいけません)。

次に示すプログラムは、`c` の値が `'a'` 以上でかつ `'z'` 以下、つまりアルファベットの小文字の範囲にはいっていれば「OK」と表示します。

```
#include <stdio.h>
main()
{
    char c;

    setbuf(stdout, NULL); .....MSX-C Ver.1.2にのみ必要
    puts("a から z までのキーを入力して下さい：");
    c = getche();

    if (c >= 'a' && c <= 'z')
        puts("\nOK");
}
```

図 6.9 「&&」演算子の使用例

● 「||」演算子 —— または～なら(論理 OR)

「`x || y`」は、`x` が成立しているか、または `y` が成立しているということを表します(これもやはり「|」と「|」の間に空白を入れてはいけません)。

次に示すのは、「||」演算子を使って、`y` か `Y` のキーが押されたときに「OK」と表示するプログラムです。

```

#include <stdio.h>
main()
{
    char c;

    setbuf(stdout, NULL); .....MSX-C Ver.1.2にのみ必要
    puts("ヨロイ テ'スル (y/n) ? ");
    c = getche();

    if (c == 'y' || c == 'Y') .....「yまたはYならば」という条件を表す
        puts("YnOK");
}

```

図 6.10 「||」演算子の使用例

● 「!」演算子 —— ～でない場合は(論理 NOT)

「!x」は、xが成立しなければ、という条件を表します。次のプログラムは、押されたキーがnでもNでもなければ「OK」と表示します。

```

#include <stdio.h>
main()
{
    char c;

    setbuf(stdout, NULL); .....MSX-C Ver.1.2にのみ必要
    puts("ヨロイ テ'スル (y/n) ? ");
    c = getche();

    if (!(c == 'n' || c == 'N')) .....「(nまたはN)でなければ、
        puts("YnOK");                .....という条件を表す
}

```

図 6.11 「!」演算子の使用例

■ 代入した値を判定する —— 代入式

次にここで、Cの条件判断の特徴である「代入式」の利用法について説明しておきましょう。これは効率的なプログラミングの道具でもあり、またプログラムをわかりにくくする元凶でもあるというCに特有のユニークな存在です。

代入式とは簡単にいえば、代入文から最後のセミコロンを取り除いたものです。つまり、

```
x = 1;
```

という代入文からセミコロンを取り除けば、

```
x = 1
```

という代入式になるわけです。代入式は変数に値を代入した後、その値を代入式自身の値として利用できます。これが最もよく利用されるのは、図6.12のプログラムに示すように、変数に代入した値がある条件を満たすかどうか、すぐに調べる場合です。

```
#include <stdio.h>
main()
{
    char c;

    if ((c = getche()) == 'y')
        puts("OK");
}
```

図 6.12 変数に代入した値をそのまま利用する

このプログラムのif文は、まず変数cにgetche()で1文字入力し、その入力した文字がyであれば「OK」と表示します。

63 繰り返し

以上で if 文を使った条件判断の説明は終わりにして、ここから繰り返し処理のプログラミングについて述べることにします。

■ 回数を指定した繰り返し —— for 文

C のプログラムでループを作る方法には、いくつか種類がありますが、一番なじみやすいのは、BASIC の FOR～NEXT 命令とよく似た for 文でしょう。C の for 文の書式は次のとおりです。

for (初期設定 ; 実行条件 ; 再設定) 文

この for 文を実行すると、最初に「初期設定」を 1 回だけ行い、そして「実行条件」が成立している間「文」を実行しては「再設定」を繰り返します。この動作をフローチャートで表せば、図 6.13 のようになります。

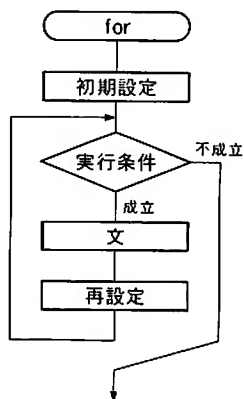


図 6.13 for 文のフローチャート

図 6.14 のプログラムに for 文の実例を示します。これは、 $i=1$ から始めて、 $i \leq 10$ である間、 i を 1 ずつ増やしながらループするわけですから、結局文字 A が 10 個表示されることになります。

```
#include <stdio.h>
main()
{
    int i;

    for (i = 1; i <= 10; i = i + 1)
        putchar('A');
}
```

AAAAAAAAA ← 実行結果

図 6.14 for 文を使って putchar() を 10 回実行する

このように、ある変数を初期値から終値まで一定の値を加えながら繰り返すという動作は、for 文の最も基本的な使い方といってよいでしょう。これは、BASIC の FOR~NEXT 命令とまったく同じ動作でもあります。実際に、

FOR I=初期値 TO 終値 STEP 増減値 ……

という BASIC の FOR~NEXT 命令は、次の for 文に機械的に置き換えることができます(変数の値が増加するか減少するかによって、条件判定の不等号の向きが異なるところに注意してください)。

for (i=初期値; i<=終値; i=i+増加値) ……

for (i=初期値; i>=終値; i=i-減少値) ……

しかし C の for 文は、このような単純な用途以外に、もっといろいろな使い方ができるところに大きな特徴があります。たとえば、for 文のループ変数は、値を足していけるだけではありません。図 6.15 のプログラムは、 i の値を 1 から始めて 2, 4, 8, ……と 2 倍しながら 256 まで増加させることを意味しています。

```

#include <stdio.h>
main()
{
    int i;

    for (i = 1; i <= 256; i = i * 2) {
        printf("%d", i);
        putchar(' ');
    }

    1 2 4 8 16 32 64 128 256 ← 実行結果

```

図 6.15 for 文のループ変数を倍加させる

また次のプログラムは、まず c に 'n' を代入しておいて、c が 'y' に等しくなるまで 1 文字入力が続けることを意味します。これなどは C の for 文としては正しいものですが、もう BASIC の FOR~NEXT 命令からの類推では動作を理解できないかもしれませんね。

```

#include <stdio.h>
main()
{
    char c;

    for (c = 'n'; c != 'y'; c = getche())
        puts("OK ｼﾞｽｶ (y/n) ?");

    puts("ﾌﾚﾊﾞ ｵｶｯﾀ");
}

```

↓ 実行結果

```

OK ｼﾞｽｶ (y/n) ?
n
OK ｼﾞｽｶ (y/n) ?
N
OK ｼﾞｽｶ (y/n) ?
y
ﾌﾚﾊﾞ ｵｶｯﾀ

```

図 6.16 自由度の高い for 文の書式

要するに、Cのfor文では「初期設定」と「実行条件」と「再設定」の組み合わせいかんで、どんなループ条件でも表せるようになっているのです。はじめのうちは、あまり妙な条件を設定するのも考えものですが、for文は非常に便利な使い方ができるということだけは覚えておいてください。

■ 空ループによる時間かせぎ —— 空文

BASICでは、単なる時間かせぎのために、

```
FOR I=1 TO 1000: NEXT
```

という何もしないループ(空ループ)を使うことができました。Cのfor文で同様なことを行うには、

```
for (i=1; i<=1000; i=i+1)
    ;
```

と書きます。このfor文は、繰り返しの本体に、ただひとつのセミコロンが置かれています。Cではこうやって何も実行しない文を表現します。このセミコロンだけの文を、とくに「空文(くうぶん)」と呼びます。

図6.17のプログラムに空文の利用例を示しました。ここでは空文を6000回繰り返していますが、普通のMSXマシンならば、これで約0.2秒の時間待ちとなります。

```
#include "stdio.h"
main()
{
    int i, t;

    setbuf(stdout, NULL); .....MSX-C Ver.1.2にのみ必要

    for (i = 1; i <= 1000; i = i + 1) {
        putchar('A');
        for (t = 1; t <= 6000; t = t + 1)
            ;
    }
}
```

6000回の空ループ
(0.2秒)

図 6.17 for 文を使った空ループ

■ 省エネ方式の代入文 —— 代入演算子

ここまで、for 文のループ変数を再設定する部分は、「 $i=i+1$ 」や「 $i=i*2$ 」などのように、ごく普通の書き方を使ってきましたが、C ではこういった変数の値の更新には表 6.4 のような「代入演算子」を利用することもできます。というより、こう書くほうが C の世界では一般的です。

演算子	意 味
$x += \text{数値}$	$x = x + \text{数値}$
$x -= \text{数値}$	$x = x - \text{数値}$
$x *= \text{数値}$	$x = x * \text{数値}$
$x /= \text{数値}$	$x = x / \text{数値}$

表 6.4 代入演算子一覧

たとえば、変数 x の値を 5 だけ増加させるなら「 $x+=5$ 」、 x の値を 3 倍にしたければ「 $x*=3$ 」という表現になるわけです。

さらに、プログラムの中でとくに使用する機会が多い「変数に 1 を加える操作」と「変数から 1 を減ずる操作」は、表 6.5 のようにもっと簡単に表現することもできます。この「++」をインクリメント演算子、「--」をデクリメント演算子と呼びます。

演算子	意 味
$++x$	$x = x + 1$
$--x$	$x = x - 1$

表 6.5 インクリメント演算子とデクリメント演算子

これらの演算子を使うと、for 文は図 6.18 のように、少したけ簡潔に表現できることになります。

```

#include <stdio.h>
main()
{
    int i;

    for (i = 1; i <= 10; ++i) { ..... iを1から10まで1ずつ増やしていく
        printf("%d", i);
        putchar(' ');
    }
    putchar('\n');

    for (i = 1; i <= 2187; i *= 3) { ..... iを1から2187まで3倍ずつ増やしていく
        printf("%d", i);
        putchar(' ');
    }
    putchar('\n');
}

```

図 6.18 for 文の条件を簡潔に書く

なお、代入演算子やインクリメント／デクリメント演算子は、for 文専用というわけではなく、プログラムのどこで使ってもかまいません。そしてまた、これらの演算子を工夫して使うと、非常に効率のよいプログラムを書くことができます。それらの実例もおいおい紹介していきましょう。

■ 条件が成立するまで繰り返す —— while 文と do 文

さて、ここまで紹介してきた for 文は、原則としてループ変数でコントロールされる動作を扱うものでした。しかし、通常のプログラミングで出てくるループは、特定のキーが押されるまで繰り返すとか、ファイルを最後まで読み込むとか、ループ変数を持たないものも少なくありません。そのような繰り返し処理のために用意されたものが、while 文と do 文です。

まず、while 文の書式を一般的な形で示すと次のようになります。

while (条件) 文

これは、「条件」が成立している間、「文」を繰り返すという意味になります。図 6.19 は while 文の動作をフローチャートで表したものです。

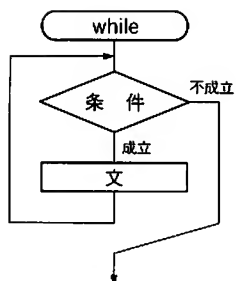


図 6.19 while 文の動作を表すフローチャート

次のプログラムに while 文の実例を示します。このプログラムは、'q'のキーが押されるまで入力した文字を画面に表示し続けます。

```

#include <stdio.h>
main()
{
    char c;

    setbuf(stdout, NULL); .....MSX-C Ver.1.2にのみ必要

    while ((c = getch()) != 'q')
        putchar(c);
}

```

図 6.20 while 文を使ったループ

while 文はループの最初で条件の判定を行うものですが、場合によっては、何かある動作を行った結果でループを続行するかどうかを決めたほうが都合のよいこともあります。そのために用意されたものが do 文です。

do 文の書式は次に示すとおりです。

do 文 while (条件);

この do 文は、まず「文」を実行してから「条件」を判断し、それが成立していればループを繰り返します。図 6.21 はこの動作をフローチャートで表したものです。

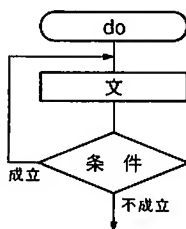


図 6.21 do 文のフローチャート

なお do 文の書式は本来は上に示したとおりなのですが、実行する文が 1 個だけの場合、プログラムは、

```
do
    文
while (条件);
```

という形で書かれることになり、「while」を while 文の始まりと見誤りやすくなって好ましくありません。

そこで一般には、do 文は実行文が 1 個だけでもかならずブロックに閉じ込めて、

```
do {
    文
} while (条件);
```

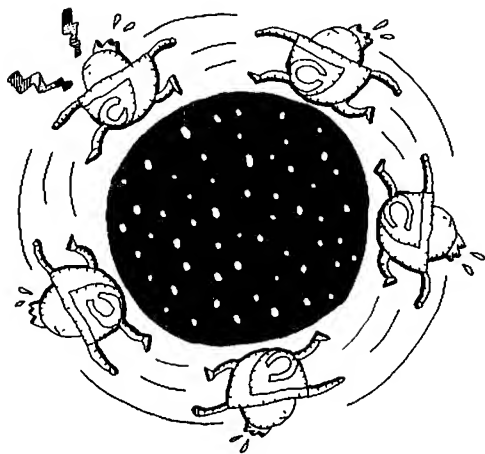
という書き方をします。つまり do 文はかならず「do {」で始まり、「} while」で終わるものと考えてしまうわけです。本書でもこの方法をとることにしました。

図 6.22 に示すプログラムは、do 文を使った実例です。ここでは、数字の 1 から 5 までのどれかのキーが押されるまで 1 文字入力を繰り返しています。

```
#include <stdio.h>
main()
{
    char c;

    do {
        puts("1 から 5 までの数字を入力してください");
        c = getche();
        putchar('\n');
    } while (c < '1' || c > '5');
}
```

図 6.22 まず「文」を実行してから条件判断をする do 文



64

それ以外の制御構造

ここまで紹介した制御文のほかに、Cにはループ文の補助的な存在である `break` 文と `continue` 文、そして場合わけをするのに便利な `switch` 文の3つが用意されています。

■ ループから抜け出す —— `break` 文

`break` 文は、次のような書式を持っています。

```
break ;
```

`break` 文は基本的にループの中で実行してはじめて意味を持つ文です。この文を実行すると、ループを強制的に中断して途中から抜け出すことができます(図 6.23)。

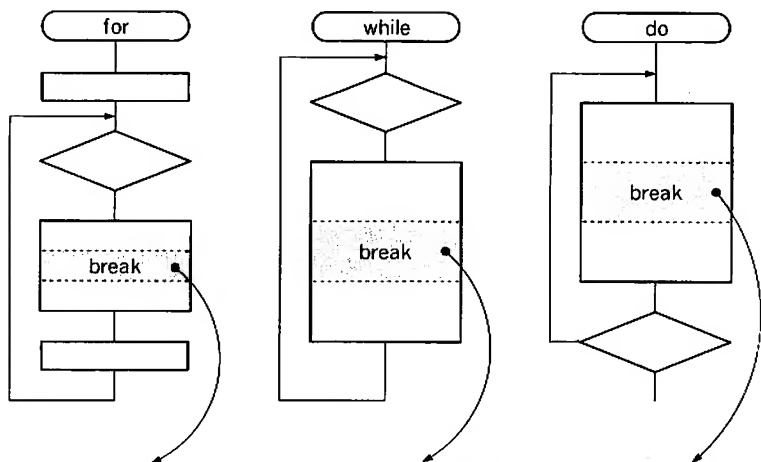


図 6.23 `break` 文の動作

ループの処理を1回だけスキップする —— continue 文

break 文と少し似ているものに、continue 文があります。書式は以下のとおりです。

```
continue ;
```

continue 文もループの中で使ってはじめて意味を持つ文ですが、こちらはループから完全に抜け出してしまうのではなく、その回のループ本体の実行を1回分だけキャンセルする働きがあります。つまり、continue 文があると処理はそこで止まり、次の繰り返しの進みます。この動作は図 6.25 のように表せます。

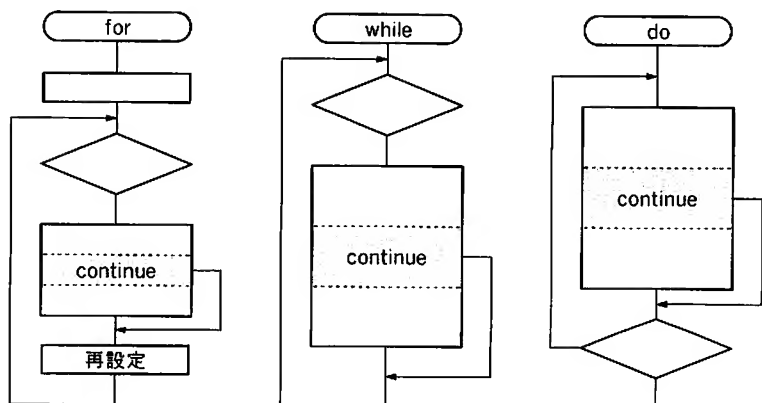


図 6.25 continue 文の動作(各ループ文に対する3種)

図 6.26 に示すプログラムは、さきほどのプログラムの break 文を continue 文に書き換えたものですが、こちらは q という文字を入力してもループは中断しません。ただループの中で continue 文以降にある部分がスキップされるだけとなっています。

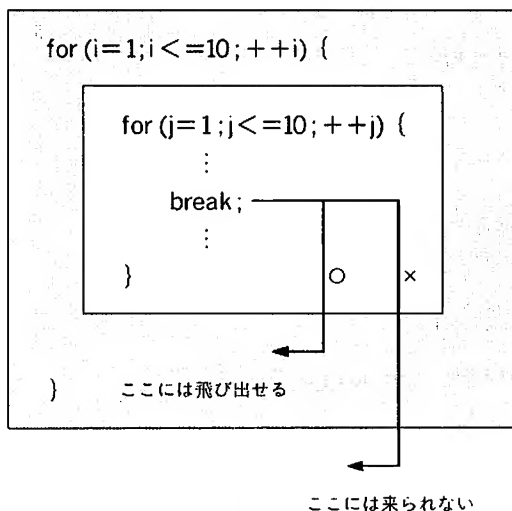


図 6.27 2 重ループからの脱出

そこで用意されたプログラム制御文が goto 文です。goto 文は適当なラベル付きの文に、無条件でプログラムの流れを移すことができます。ラベル付きの文と goto 文の書式は、それぞれ以下のとおりです。

ラベル名： 文

ラベル名は goto 文の飛び先を示す目印で、変数名と同様にアルファベットの大文字と小文字、およびアンダースコア記号の組み合わせであれば、自由に名付けてかまいません。

なんらかの文の前にラベル名とコロン記号「:」を置いたラベル付きの文は、goto 文の飛び先として指定できます。なお、これはあくまでも「ラベル付きの文」でありますから、ラベルの後にはかならず文(空文でもよい)が必要です。

goto ラベル名;

そして goto 文は、指定されたラベル付きの文にジャンプします。これは飛び先がどこにあってもかまいません(注:ただし同じ関数の中であること)。

ですから多重の入れ子になったループの内側から、一番外側まで抜け出すことも簡単です。

図 6.28 に 2 重のループの中から goto 文によっていっきに脱出するプログラムの例を示します。なお、ここで使っている kbhit() は、その時点でキーが押されていれば真、押されていなければ偽の論理値を返すライブラリ関数です。

```
#include <stdio.h>
main()
{
    int i, j;

    for (i = 1; i <= 100; ++i) {
        for (j = 1; j <= 100; ++j) {
            puts("%ni + j = ");
            printf("%d", i + j);
            if (kbhit()) {
                puts("%nBREAK!%n");
                goto exit_all;
            }
        }
    }
    exit_all:
    puts("end");
}
```

図 6.28 goto 文の使用例

■ 効果的な場合わけ処理 —— switch 文

さて最後に残ったのは、C の制御構造の中でも最もクセが強く扱いにくく、それだけにイザというとき頼りになる switch 文です。これは BASIC というならば ON~GOTO 命令に相当する「多重分岐」のための制御文です。switch 文の書式は次のとおりです。

1

```

switch (式) {
    case 定数 1: 文 1-1 文 1-2 .....
    case 定数 2: 文 2-1 文 2-2 .....
                :
    default:    文 n-1 文 n-2 .....
}

```

switch 文は、まず「式」の値を計算します。そしてその値と「case 定数:」の形で書かれた定数値を上から比較していき、両者が等しくなった部分以降の文をすべて実行します。式の値がどの定数とも一致しなければ、「default:」以降の文を実行します。

この動作は、あれこれ説明するよりも実例を見ていただくほうがわかりやすいかもしれません。図 6.29 に示すのは、switch 文の動作を見るための簡単なプログラムです。

```

#include <stdio.h>
main()
{
    switch (getche()) {
        case 'a':
            puts("\nApple");
        case 'b':
            puts("\nBanana");
        default:
            puts("\nNot found");
    }
}

```

実行結果 1

```

a ..... a を入力した
Apple ..... 「case 'a':」以降の文がすべて実行される
Banana
Not found

```

実行結果 2

```

b ..... b を入力した
Banana ..... 「case 'b':」以降の文がすべて実行される。
Not found

```

図 6.29 switch 文の動作を見るプログラム

さて、この結果からもわかるとおり、与えられた式の値がある case に一致すると、switch 文はそれ以降の case に相当する文をすべて実行してしまいます。

しかし普通は a が入力されたら A の動作、b が入力されたら B の動作というように、それぞれ別々の動作を行う機能がむしろ要求されるでしょう。そのためには、図 6.30 のプログラムに示すように、switch 文の中で break 文を利用します。

```
#include <stdio.h>
main()
{
    switch (getche()) {
        case 'a':
            puts("Apple");
            break;
        case 'b':
            puts("Banana");
            break;
        default:
            puts("Not found");
            break;
    }
}
```

実行結果 1

```
a ..... a を入力した
Apple ..... Apple だけ表示された
```

実行結果 2

```
b ..... b を入力した
Banana ..... Banana だけ表示された
```

図 6.30 break 文を用いた一般的な switch 文の使用例

break 文でループから抜け出せることについては説明しましたが、このように switch 文から抜け出すにも break 文が利用できるわけです。

ところで、なぜ switch 文がこんなややこしい構造になっているかといいますと、それは図 6.31 のプログラムのような使い方、つまり複数のラベルが同じひとつの飛び先を指定できるように考えたからということです。

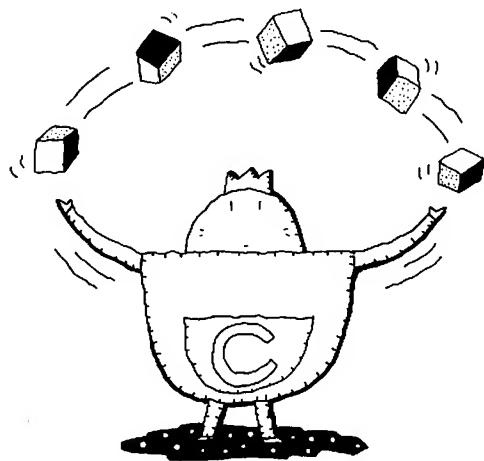
```
#include <stdio.h>
main()
{
    switch (getche()) {
        case 'A':
        case 'a':
            puts("Apple\n");
            break;
        case 'B':
        case 'b':
            puts("Banana\n");
            break;
        default:
            puts("Not found\n");
            break;
    }
}
```

図 6.31 大文字も小文字も同じ飛び先に飛ぶ

ともあれ、switch 文には break 文、これはもうワンセットで覚えてしまったほうがよいかもしれませんね。

7章

数値データの扱い方



ここまでの章では、数なんて計算できてあたりまえ、じつたって足し算や引き算するのに BASIC と変わるわけないよーん、という安直な姿勢で説明を進めてきました。まあ確かにプログラミング言語が違ってても、基本的な数の扱いにそれほど差が出るはずもありません。1+1 が 2 になるのは宇宙の真理です。

しかし、すでに見てきたとおり、ちょっとプログラムの奥の細道をのぞいてみると、C と BASIC はアレコレと違う振舞いをします。数値データの扱いにしてもしかり、単なる数値といって油断はできません。

というわけで、この章では、C 言語における数値データの扱いについて、しっかりと基礎を固めておきましょう。



71

数値のいろいろな書き表し方

プログラミングの場では、10進数以外の数値表現が使えると便利な場合があります。とくにグラフィックパターンを表示したり、ハードウェアを直接操作するような場合には、16進数は欠かすことができません。

ここでは、それらのいろいろな数値表現と、その画面への出力方法について説明します。

4 種類の数値表現

MSX-Cに用意されている数値の表現方法を順にあげてみると、まずごく普通の10進数、そして16進数と8進数があります。また、文字定数も一種の数値と考えてよいでしょう。結局のところMSX-Cでは、表7.1に示した4種類の数値表現が利用できることになります。

数値表現	例	表記の方法	データの型
10進数	-1536	ごく普通の数の表記	int/unsigned
16進数	0xf a00	先頭に0xを付ける	unsigned
8進数	0175000	先頭に0を付ける	unsigned
文字定数	'A'	文字をシングルクォートで囲む	char

表 7.1 Cの数値表現

● 10進数

10進数とは、いまさら言うまでもありませんが、普通の10進数字(0~9)を並べたものです。ただしCでは、数字の先頭に0(ゼロ)を付けると、後で述べる8進数とみなされてしまいますから注意してください。

なお、10進数にはint型とunsigned型の2種類の解釈のしかたがありますが、このことについては次の7.2節で説明します。

● 16 進数

ハードウェアを直接操作したり、データのビットパターンを見るようなテクニカルな用途に使う数値表現といえば、この 16 進数で決まりです。

C で 16 進数を表すには、0x(ゼロエックス)に続けて、16 進数字(0~9, a~f)を並べます。このとき 16 進数字の a~f は大文字の A~F でもかまいませんが、0x の x はかならず小文字で書かなくてはなりません。

● 8 進数

いま、テクニカルな用途に使う数値表現といえば 16 進数だ！ と言い切ってしまいましたが、実はそう言えるようになったのは最近のことです。一昔前の(C 言語が誕生した)時代には、16 進数よりむしろ 8 進数のほうがよく利用されていました。その名残からか、C では今でも 8 進数が使われることが少なくありません。

8 進数を表すには、「0(ゼロ)」に続けて 8 進数字(0~7)を並べます。これは一見すると 10 進数と間違えやすいので注意してください。

● 文字定数

文字定数はシングルクォート記号で 1 個の文字を囲んだものです。文字定数が数値の表現だといわれると、すこし奇妙な感じですが、C では文字定数はその文字コードに等しい数値として扱ってかまいません。

これらの各形式で書き表した数値は、10 進数とまったく同様にプログラムの中で利用することができます。図 7.1 に、16 進数/8 進数/文字定数を使ったサンプルプログラムを示します。

なお、文字定数は char 型のデータですから、printf()に直接、

```
printf("%d", 'A');          …… 誤り
```

という形で表示させることはできません。一般の C ではこれでも問題ないのですが、関数のパラメータの型の区別に厳しい MSX-C では、図 7.1 の 7 行目のように int 型にキャスト(後述)する必要があります。

```

0: #include <stdio.h>
1: main()
2: {
3:     int i;
4:
5:     printf("%d", 0x123);      putchar('\n'); .....16進数を10進数表示
6:     printf("%d", 0123);      putchar('\n'); ..... 8進数を10進数表示
7:     printf("%d", (int)'A');  putchar('\n'); .....文字定数を10進数表示
8:
9:     i = 0x123 + 0123 + 'A';
10:    printf("%d", i);
11: }

```

実行結果

```

291 .....16進数 0x123は10進数に直すと291
83  ..... 8進数0123は10進数に直すと83
65  .....文字定数'A'はAの文字コードに相当する65
439 ..... 0x123+0123+'A'は10進数に直すと439

```

図 7.1 16 進数, 8 進数, 文字定数の使用例

16 進数や 8 進数の表示

さきほどの図 7.1 に示したプログラムでは、10 進数以外の形式で表した数値でも、`printf()` で画面に出力するときには 10 進数として表示されました。これは別におかしな現象ではありません。なぜなら 16 進数や 8 進数という特別な「数」が存在しているわけではないからです。

16 進数や 8 進数などは、あくまでもプログラムリスト上での数値の書き方にすぎません。そして数値がどのように書かれていても、コンパイルしてしまえば結果は一緒、コンピュータの内部では、10 進数も 16 進数も 8 進数も、みんな 2 進データに変換されています。

そのため `printf()` を使うときは、数値データをどの形式で画面に表示したいのか、こちらから指定してやらなくてはなりません。逆にいえば、指定の方法さえ変えれば、ある数値をどの形式でも自由に表示できるのです。

たとえば、ここまでは `printf()` にはかならず “%d” の指定記号を使ってきましたが、これは数値を 10 進数の形式で表示せよという意味でした。printf

()では、この10進数に加えて、16進数、8進数、文字、符号なし10進数と、合わせて5種類の数値形式が指定できます。

表 7.2 にそれぞれの指定記号をまとめます。

記 号	意 味
%d	10進形式
%x	16進形式
%o	8進形式
%c	文字形式
%u	符号なし10進形式

表 7.2 printf() の表示指定記号

以下、各記号の使い方を簡単に説明しましょう。

●数値を10進数として表示——%d 記号

“%d”は、すでに紹介してきたとおり、数値データを-32768～32767の範囲の10進数の形式で表示します。後で述べる“%u”との違いに注意してください。

●数値を16進数として表示——%x 記号

“%x”は、数値データを16進数として表示します。ただし、このとき16進数の先頭に0xは表示されません。またMSX-Cでは、16進表示には大文字のA～Fを使いますが、コンパイラによっては小文字のa～fを表示するものもあります。

●数値を8進数として表示——%o 記号

“%o”（oは小文字のオー）記号を使用すると、数値データは8進数の形式で表示されます。ただし8進数の先頭に0は付きません。

●数値を1個の文字として表示——%c 記号

“%c”はちょっと特殊な形式で、数値データを文字コードと考え、それに

相当する文字 1 個を表示します。

これは putchar() の動作によく似ていますが、putchar() が char 型のデータを表示するのに対し、この printf() の %c 指定は、int 型(または unsigned 型)のデータを文字として表示するものです。

●数値を符号なし 10 進数として表示——%u 記号

"%u" は、数値データを 0 ~ 65535 の範囲の正の 10 進数として表示します。たとえば 0xffff という 16 進数は、%d 指定ならば -1 と表示されますが、%u 指定を使うと 65535 になります。これは後で述べる unsigned 型のデータを表示するために用意された表示形式です。

図 7.2 に 5 種類の形式で 64 から 80 までの数値を表示するプログラムを示します。

```
#include <stdio.h>
main()
{
    int i;

    for (i = 64; i <= 80; ++i) {
        printf("%d", i);    putchar(' ');
        printf("%x", i);    putchar(' ');
        printf("%o", i);    putchar(' ');
        printf("%c", i);    putchar(' ');
        printf("%u", i);    putchar('\n');
    }
}
```

```
64 40 100 0 64
65 41 101 A 65
66 42 102 B 66 ← 実行結果
: : : :
79 4F 117 O 79
80 50 120 P 80
```

図 7.2 printf() による 5 種類の形式の数値表示

7 2

数値データの型と、 その利用法

Cで扱う数値データは、ある決まった「データの型」を持っています。これは基本的には扱う数値範囲の違いを区別するのですが、数値の代入や比較演算を行う際にも、データ型の違いはプログラムにさまざまな影響を与えます。

この節では、そのデータ型とはどういうものか、さらに個々のデータ型の特徴について見ていきましょう。

データの型とは何か

最初に簡単な実験を1つ。まず、16進数の0x9000と0x3000という2つの数を考えてください。これらを10進数形式で表示してみると、図7.3のように、0x9000は-28672というマイナスの数値、0x3000は12288というプラスの数値になっています。

```
#include <stdio.h>
main()
{
    puts("\n0x9000 == ");
    printf("%d", 0x9000);
    puts("\n0x3000 == ");
    printf("%d", 0x3000);
}

0x9000 == -28672
0x3000 == 12288
```

← 実行結果

図 7.3 0x9000 と 0x3000 の値を確認する

さてそこで、この2つの16進数の大小を比較し、どちらが大きいかを表示させてみます。図7.4がその実行結果です。

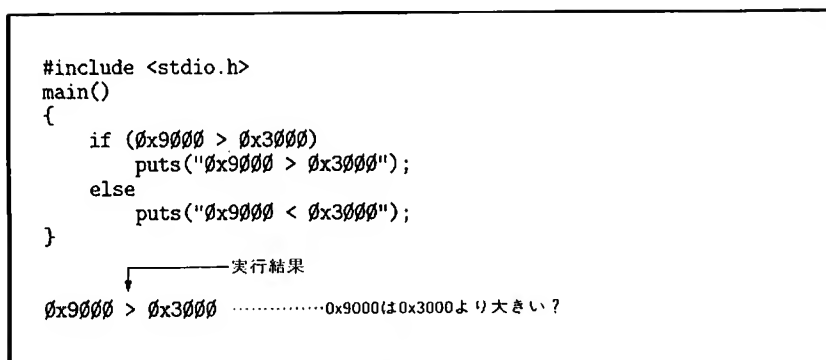


図7.4 この結果は変ではないか？

すると結果は上のように、「0x9000 > 0x3000」と表示されたと思います。マイナスの数であるはずの0x9000のほうが、プラスの数であるはずの0x3000よりも大なりとは、これいかに。

その理由は、Cでは16進数をunsigned型のデータとして扱う約束になっているからです。unsigned型とは、別名「符号なし整数」とも呼ばれるもので、int型のデータとは基本的に種類の異なるデータ型とみなされます。

●データの型とはデータを扱う考え方である

MSX-Cでは整数を16ビットの2進データで表現しますが、この2進データは符号付きの整数であるとも考えることも、符号なしの整数と考えることもできます。

1001000000000000	符号付き整数と考えれば-28672
(16進では0x9000)	符号なし整数と考えれば36864

そしてCでは、データを符号付きの整数として扱う場合、それをint型のデータと呼び、符号なしの整数として扱う場合、それをunsigned型のデータと呼ぶのです。

unsigned 型と int 型のデータの違いの例を1つ図 7.5 に示します。ここでは、さきほどの 0x9000 と 0x3000 を、それぞれのデータ型の変数に代入した上で比較しています。

```
#include <stdio.h>
main()
{
    int      i1, i2; .....int 型の変数 i1 と i2 を用意
    unsigned u1, u2; .....unsigned 型の変数 u1 と u2 を用意

    i1 = 0x9000;
    i2 = 0x3000;
    if (i1 > i2) .....unsigned 型で比較する
        puts("int ----- 0x9000 > 0x3000\n");
    else
        puts("int ----- 0x9000 < 0x3000\n");

    u1 = 0x9000;
    u2 = 0x3000;
    if (u1 > u2)
        puts("unsigned -- 0x9000 > 0x3000\n");
    else
        puts("unsigned -- 0x9000 > 0x3000\n");
}

int ----- 0x9000 < 0x3000
unsigned -- 0x9000 > 0x3000 ← 実行結果
```

図 7.5 int 型の比較と unsigned 型の比較の違い

結果はごらんのとおり、同じ 2 つの数値でも、int 型では「0x9000 < 0x3000」、unsigned 型では「0x9000 > 0x3000」とみごとに違う回答が得られました。このように、コンピュータが数値をどのようなものと考え、どのように扱うかを区別するのが「データの型」なのです。

3 種類のデータの型

ここで、MSX-C に用意されている、基本的な数値データの型を紹介しておきましょう。表 7.3 にその一覧を示します。

データ型	扱う数値の範囲	サイズ
int型	-32768 ~ 32767	16ビット(2バイト)
unsigned型	0 ~ 65535	16ビット(2バイト)
char型	0 ~ 255	8ビット(1バイト)

表 7.3 MSX-C の数値データの型

他のCコンパイラでは、これら以外にも小数点以下まで表現できる実数型(float型、double型)や、約20億までの大きな整数が表現できる倍精度整数型(long型)が利用できる場合もありますが、MSX-Cには上の3種類のデータ型しか用意されていません(ただし、別売の「MSX-C ライブラリ」を併用すると、MSX-Cでも実数や倍精度整数が使えるようになります)。

● int 型のデータ

int型のデータは、MSX-Cで使われる最も基本的な数値データです。これはビット数でいうと16ビットのサイズを持ち、そのためint型のデータを変数に記憶させる場合には2バイトのメモリを必要とします。

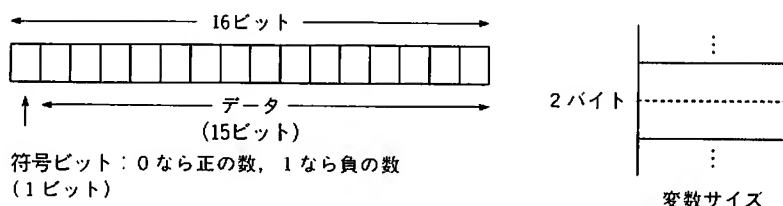


図 7.6 int 型のデータ構造

int 型のデータには以下のものがあります。

- ・ -32767~32768 の範囲の 10 進定数
- ・ int 型の変数
- ・ (int) 演算子でキャストした任意のデータ

3 番目にあげた「(int)」は、キャスト演算子と呼ばれるものの1つで、型

の名前をカッコで囲むことにより表されます。キャスト演算子を頭に付けると、本来はほかの型であるデータを、指定した型のデータとして強制的に扱うことができます。

たとえば、16 進数は本来は unsigned 型のデータなのですが、図 7.7 のようにキャスト演算子(int)を使用すると、コンパイラはこれを int 型のデータとして扱ってくれます。なお、このようにキャスト演算子を使ってデータの型を変換することを、そのデータ型に「キャストする」といいます。

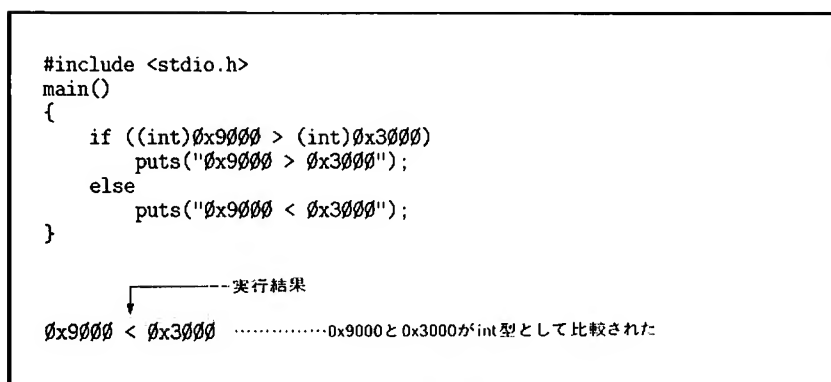


図 7.7 キャスト演算子(int)の効果

● unsigned 型のデータ

unsigned 型は、すでに述べたように、int 型と同じ 16 ビット (2 バイト) の数値データを符号なしの整数として扱うためのデータ型です。

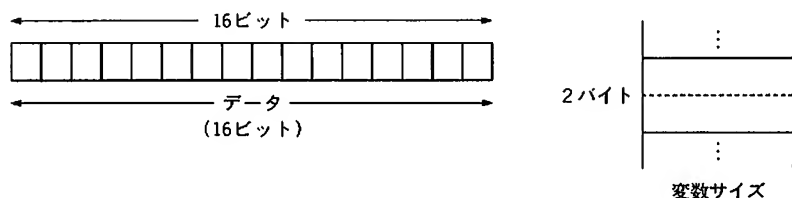


図 7.8 unsigned 型のデータ構造

unsigned 型のデータには以下のものがあります。

- ・ 32768 以上の 10 進定数(32768~65535)
- ・ 16 進定数
- ・ 8 進定数
- ・ unsigned 型の変数
- ・ (unsigned)演算子でキャストした任意のデータ

unsigned 型のデータを利用したい場合は、かならず変数も unsigned 型として宣言してください。int 型と unsigned 型が混乱すると、わけのわからない結果が出てきて何時間も頭をひねることがあります。試しに図 7.9 のプログラムを実行してみてください。

```
#include <stdio.h>
main()
{
    int i;

    for (i = 0; i <= 60000; ++i) {
        printf("%d", i);
        putchar(' ');
    }
}
```

実行結果
何もおきない

図 7.9 int 型の変数で unsigned 型の 60000 はカウントできない

このプログラムは、0 から 60000 までの数値を表示させるつもりだったのですが、いざ実行してみると、なにもしないで終了してしまいました。その原因は、ループ変数 *i* が int 型として宣言されていたからです。

MSX-C が異なる型のデータを比較する場合、変数と定数ではつねに変数のデータ型のほうが優先されます。ですから、この場合「*i* <= 60000」という条件判断は int 型データの比較とみなされ、上の for 文はコンパイラに次のように解釈されてしまいます。

```
for (i = 0; i <= -5536; ++i) {
    .....    (60000 を int 型に直すと -5536)
```

これは、変数 *i* の値を 0 から増加させつつ -5536 まで繰り返すループ、いわゆる「おまえはすでに終わっている」ループですから、当然何も表示されないわけです。

数値をこのように unsigned 型のデータとして扱いたい場合には、変数も次のように unsigned 型として宣言しなければなりません。

```
unsigned i;          ← unsigned 型として宣言する
```

● char 型のデータ

char 型は、数値を 8 ビット (1 バイト) サイズの符号なし整数として扱うためのデータ型です。

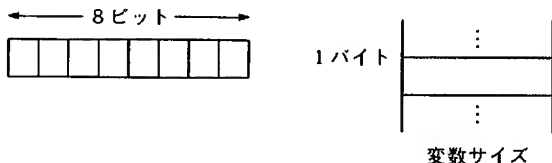


図 7.10 char 型のデータ構造

char 型のデータには以下のものがあります。

- ・ 文字定数
- ・ char 型の変数
- ・ (char) 演算子でキャストした任意のデータ

これは MSX-C に用意されたデータ型の中で、ただ 1 つデータの内部表現のサイズが異なるため (ほかのデータ型はすべて 2 バイトです)、利用にあたっては注意が必要です。

たとえば putchar() は char 型データ専用の表示関数ですから、これで int

型のデータを表示しようとしてもうまくいきません。

```
putchar(65);           …… 誤り(65 は int 型)
putchar('A'+1);       …… 誤り('A'+1 は int 型)
```

その逆に printf() は int 型のデータしか扱えませんから、次のように printf() に char 型の文字定数を与えても、望んだ結果は得られません。

```
printf("%d", 'A');     …… 誤り('A' は char 型)
```

これらの例のように、要求されているものと異なる型のデータを使う場合には、キャスト演算子を使ってデータの型を目的の型に強制的に合わせる必要があります。上の 3 つの誤った関数呼び出しを正しいデータ型にキャストすると、次のようになります。

```
putchar((char)65);     …… 正しい(文字 A が表示される)
putchar((char)('A'+1)); …… 正しい(文字 B が表示される)
printf("%d", (int)'A'); …… 正しい(数値 65 が表示される)
```

MSX-C ではデータの型の間に表 7.4 に示す優先順位があり、数式のデータ型は、その中に含まれる一番優先順位の高いデータに合わされます。

強い ↑ ↓ 弱い	unsigned型の変数 int型の変数 char型の変数 unsigned型の定数(16進数, 8進数, 32768以上の10進数) int型の定数(-32767~32768の範囲の10進数) char型の定数(文字定数)
--------------------	---

表 7.4 データの型の優先順位

簡単にいうと、定数よりも変数のほうが強く、char 型よりも int 型、int 型よりも unsigned 型のほうが強という法則があるわけですが、いちいち考えるのが面倒ならば、データの型が混合していて結果がアヤシくなりそうな場合には、かならずキャスト演算子を使ってしまうというのも手かもしれません。

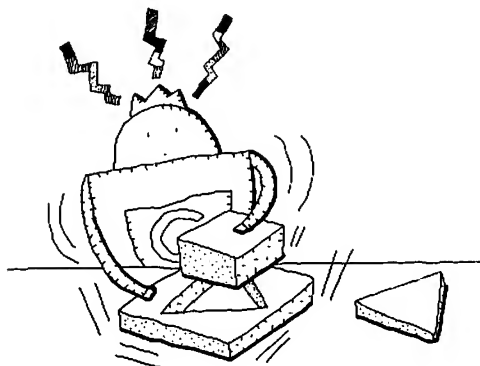
■ データ型の整合性チェック

ここでは、前項の説明を受けて、データの型の誤りから起こるバグを未然に防ぐ方法について述べます。これはデータの型を正しく扱っている限り無用の知識ですから、もし読むのが面倒ならば読みとばしていただいてもかまわないでしょう。

MSX-C では関数に引き渡すパラメータのデータ型の区別に厳しく、これを間違えると正常な動作は望めません。しかしそれにも関わらず、MSX-C コンパイラは関数パラメータのデータ型の誤りに対してはエラーチェックを行ってくれません。

そのため、コンパイルは正常に終了したのに、いざ実行してみると思い通りの結果が得られないという事態も起こりえます。とくに、他のCで動いているプログラムをMSX-Cに移植する場合などには、しばしばこの問題が影響してきます。一般のCではint型とchar型はほとんど違いを意識せずに使えるため、たいていの場合両者は混ぜこぜに使われているからです。

そこでMSX-Cには、FPC(ファンクション・パラメータ・チェッカ)というサポートツールが用意されました。FPC コマンドは、いま作成しているプログラム中の関数呼び出しと、別ファイルに登録してある関数の正しい使用法を照らし合わせ、誤った関数呼び出しの行われている場所をすべて示してくれます。



FPC コマンドによる関数呼び出しのエラーチェック方法を、実際にバグを含んだプログラムを例にして説明しましょう。

まず、次のプログラムを"AIUEO.C"というファイル名で作成してください。これは「ア」から「ン」までの45文字を表示するプログラムなのですが、コンパイルは正常に行われるのに、実行してみるとうまく動いてくれません(みなさんも試してみてください)。

```
#include <stdio.h>
main()
{
    int i;

    for (i = 0; i < 45; i++)
        putchar('ア'+i);
}
```

図 7.11 関数へのパラメータの型を間違えた例

こんなときはFPCの出番です。FPCでプログラムをチェックするには、いったんソースプログラムをTコードファイルに変換する必要があります。つまり、次のようにCFコマンドを実行して"AIUEO.TCO"を作るわけです。

```
A>cf aiueo ☒ ..... "AIUEO.C" をCFにかける
MSX-C ver 1.10P (parser)
Copyright (C) 1987 by ASCII Corporation
complete
A>dir aiueo.tco ☒ ..... "AIUEO.TCO" の存在を確認する
AIUEO      TCO      256 89-01-08 06:30a
          1 file 366592 byte free
A>
```

図 7.12 Tコードファイルを生成する

次に、このTコードファイルを、あらかじめ用意してある"LIB.TCO"と一緒にFPCコマンドにかけます。"LIB.TCO"は、MSX-Cの全ライブラリ関数

の正しい使用法が記録された T コード形式のファイルです。

ここで、もし何も問題がなければ、FPC は単に「complete」とだけ表示するでしょう。しかし上のプログラムの場合は、次のようなエラーメッセージが出てきます。

```
A>fpc aiueo.lib .....*AIUEO.TCO*と*LIB.TCO*を照会(拡張子は省略)
MSX C function parameter checker ver 1.10s
in <aiueo.TCO> "main" calls "putchar" : 1th argument conflict
comlete
```

図 7.13 FPC のエラーメッセージ

このメッセージを日本語に意識すると「<aiueo.TCO>というファイルの中で、main()が putchar()を呼び出しているが、その1番目の引数の型が誤っている」ということになります。そこで元のソースプログラム“AIUEO.C”を見てみると、なるほど、main()の中で putchar()に与えているパラメータ「ア+i」は、char 型ではなく int 型ではありませんか(前項で述べたように、char 型定数+int 型変数は int 型になります)。

というわけで、キャスト演算子を使って、プログラムを次のように訂正します。これをコンパイルして実行すれば、最初の意図どおりにアからンまでの45文字が表示されるはずです。メデタシめでたし。

```
#include <stdio.h>
main()
{
    int i;

    for (i = 0; i < 45; i++)
        putchar((char)('ア'+i));
}
```

↑
「ア'+i」をchar型キャストする

図 7.14 FPC のメッセージに従って修正したプログラム

FPC の話が出たついでに、このコマンドによる関数呼び出しの正誤チェックについて、もう少し述べておきます。

一般に FPC のエラーメッセージは次のような形式で出力されます。わざわざファイル名(ここでは `<aiueo.TCO>`)まで表示するのは、後で説明する「プログラムの分割コンパイル」を行う際に、どのファイルの中でエラーが発生したかを見分けるためです。

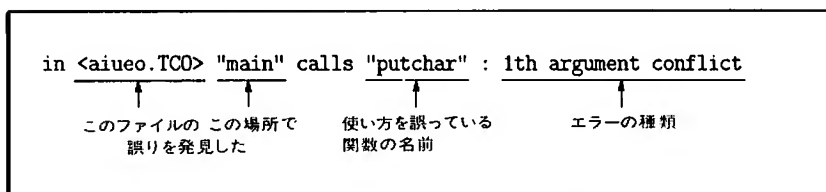


図 7.15 FPC のエラーメッセージの読み方

通常のライブラリ関数を使ったプログラムでは、FPC コマンドで発見できるエラーには全部で3種類のものがあり、それぞれ以下のようなメッセージが表示されます。

① 「…… : conflicting number of arguments」

パラメータの個数が誤っていることを示します。ただし `printf()` などの可変パラメータ関数(パラメータの個数がいくつでもよい関数)については、当然ながらこのエラーチェックは行われません。

② 「…… : ?th argument conflict」(?の部分には数字がはいる)

すでに述べたとおり、このエラーメッセージはパラメータのデータの型が誤っていることを示します。

ただし `printf()` などの可変パラメータ関数に対しては、FPC はデータ型のチェックを行いません。つまり、誤って `printf()` に `char` 型のパラメータを与えても、FPC はそれをエラーと判断してくれないのです。

ですから `printf()` には、プログラマの責任において正しい `int` 型のデータを与えてやる必要があります。まったくもって言語同断な話ですが、FPC コマンドの仕様がそうなっている以上しょうがありません。

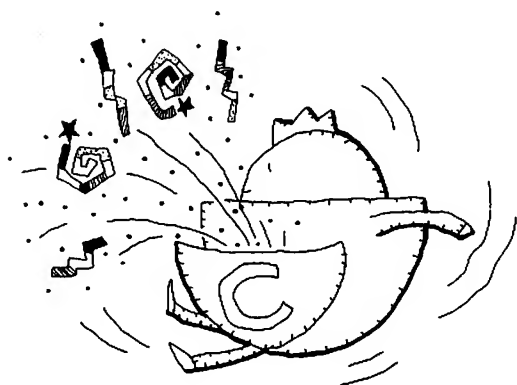
③ 「…… : undefined」

これは未定義の関数が使われていることを示します。いまの段階でこのエラーが発生するということは、MSX-Cに用意されていないライブラリ関数を使用したか、あるいは関数名のスペリングを誤ったかのどちらかでしょう。もう少し先に進むと、プログラムの分割コンパイルを行うときに、このエラーメッセージが意味を持つことになります。

以上、FPC コマンドでチェックできるエラーの中で、ライブラリ関数の使用法に関連するものを紹介しました。これ以外の FPC の機能は、それぞれ必要となった時点で改めて説明することになります。

8章

画面表示を工夫する



C のプログラミングに関する知識もだいぶ蓄積が増えました。内容が充実してきたところで、そろそろ結果の見せ方にも気をつけてみましょう。ここまでは、文字の表示にしても数値の表示にしても、単に画面に出てくれればいいやぐらいの考えでプログラムを作ってきましたが、やはりカッコよい表示を行うプログラムは、格も違って見えます。

BASIC には、CLS 命令や LOCATE 命令など、画面のコントロールを行う命令がいくつか用意されていましたが、MSX-C には、それに相当する機能のライブラリ関数はありません。しかし、コントロール文字やエスケープシーケンスを画面に出力することによって、これらの機能は簡単に実現できるのです。

81 フォーマット指定付きの数値表示

printf()を使うと、16進数や8進数などの形式で数値データを表示できることは前章で述べましたが、printf()の実力はそれだけではありません。数値の桁ぞろえや、数値以外の文字列を添える方法など、数値をさらに見やすい形式で表示する機能も用意されています。

ここでは、そのような printf() のちょっと高度な使い方を紹介することにしてしましよう。

■ 数値の桁ぞろえ表示

数値を桁ぞろえして表示する printf() の機能には、3つの種類があります。それを表 8.1 にまとめてみました。

なお、この表には 10 進形式の数値表示(%d 指定)に対する桁ぞろえの方法だけを示しましたが、これ以外の表示形式(%x, %o, %c, %u)に対しても、同様な桁ぞろえ指定が可能です。

機 能	指定記号	例	表示結果
数値の右ぞろえ	%桁数 d	printf("%6d", 123);	···· 123
数値の左ぞろえ	%-桁数 d	printf("%-6d", 123);	123····
左端を 0 で埋める	%0 桁数 d	printf("%06d", 123);	000123

表 8.1 printf() の桁ぞろえ表示機能

以下、それぞれの機能について簡単に説明します。

●数値の右そろえ表示

`printf()` で 10 進数を表示するとき、`%` と `d` の間に桁数を入れて、

%桁数 d (桁数は任意の 10 進数)

という指定を行うと、その桁数の中に数値を右そろえに表示できます。このとき、左側の余った桁は空白で埋められます。また、この右そろえ表示は、`%x`、`%o`、`%c`、`%u` の各表示形式に対しても同様に指定することができます。

```
printf("%10d", 123);
      ↓
□□□□□□□123  (←□は空白を表します)
      ← 10桁 →
```

図 8.1 右そろえの表示形式

●数値の左そろえ表示

桁数の前にさらにマイナス記号を付け、

%-桁数 d (桁数は任意の 10 進数)

という指定を行うと、数値をその桁数の中に左そろえで表示できます。これもやはり`%d`、`%x`、`%o`、`%c`、`%u` の各形式に対して指定可能です。

```
printf("%-10d", 123);
      ↓
123□□□□□□□ (←□は空白を表します)
      ← 10桁 →
```

図 8.2 左そろえの表示形式

●右そろえして左端を 0 で埋める

桁数の前に数字の 0 を付け、

%0 桁数 d (桁数は任意の 10 進数)

という指定を行った場合は、数値を右そろえにして、さらに余った左側の桁

を数字の0で埋めることができます。やはり%d, %x, %o, %c, %uのどの形式に対しても指定可能です。

```
printf("%010d", 123);    (← 010 と書いても 8 進数
                        ↓                      ではないことに注意)
0 0 0 0 0 0 0 1 2 3
←----- 10桁 ----->
```

図 8.3 右そろえて左端を0で埋める表示形式

図 8.4 に、アルファベットの文字コードを 10 進、16 進、8 進、の各形式で表示するプログラムを示します。

```
#include <stdio.h>
main()
{
    int i;

    puts(" 10進 16進 8進");
    puts("-----");

    for (i = 'A'; i <= 'Z'; ++i) {
        printf("%5c", i);
        printf("%5d", i);
        printf("%5x", i);
        printf("%5o", i);
        putchar('\n');
    }
}
```

10進	16進	8進
A	65	41
B	66	42
:	:	:
X	88	58
Y	89	59
Z	90	5A

← 実行結果

図 8.4 %d 以外の形式に対しても右そろえの指定ができる

■ フォーマット指定付きの数値表示

ここまで、`printf()`を単独の数値を表示するために用いてきましたが、実際には`printf()`にはもっと便利な使い方があります。まず、数値以外の文字と一緒に表示することができます。さらに複数の数値を1つの`printf()`で表示することもできます。ここではその2つの方法について説明しましょう。

● 表示フォーマットの指定

`printf()`の1番目のパラメータである文字列は、いままで見てきたように、数値表示のいろいろな形式(フォーマット)を指定するために使われます。そのため、この文字列は一般にフォーマット文字列と呼ばれます。

フォーマット文字列の中には、図8.5のプログラムのように、表示形式の指定(ここでは`%d`)と通常の文字を混在させることができます。

```
#include <stdio.h>
main()
{
    printf("1 ネン ハ %d ジ カン デス", 24 * 365);
}
```

1 ネン ハ 8760 ジ カン デス ← 実行結果

図 8.5 数値と文字を1つの`printf()`で一緒に表示する

この場合、`printf()`は指定記号`%d`の部分だけを表示したい数値に置き換えて、残りの文字はそのまま画面に出力します。

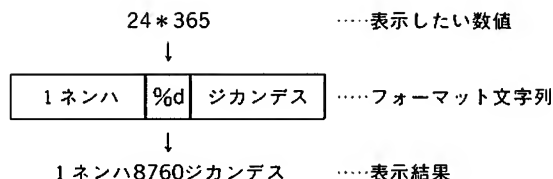


図 8.6 %d と数値の置き換えのしくみ

もちろん、`%d`だけではなく`%x`、`%o`、`%c`、`%u`の各記号や、それに桁数を加えた`%6d`や`%04x`などの指定記号も自由に使ってかまいません。

フォーマット文字列のそれ以外の部分には、`puts()`で表示する通常の文字列とまったく同じように文字を並べることができますが、ただパーセント記号`%`だけは特別です。パーセント記号は、`printf()`にとって、数値形式の指定の始まりという特殊な意味を持つため、実際に「`%`」という文字を `printf()` で表示したい場合には、図 8.7 のように記号を 2 つ重ねて `%%` と書いてください。

```
#include <stdio.h>
main()
{
    printf("ジョリト ギイハ %d %% ジョ", 3);
}
```

ジョリト ギイハ 3 % ジョ ← 実行結果

図 8.7 `printf()` で `%` という文字を表示する方法

● 複数個のデータを 1 つの `printf()` で表示する

また `printf()` は一度に複数の値を表示することも可能です。その場合は、フォーマット文字列の中に `%d` を必要なだけ並べておきます。たとえば図 8.8 のプログラムでは、39、402、 39×402 の 3 つの数を `printf()` で表示しています。

```
#include <stdio.h>
main()
{
    printf("%d * %d == %d", 39, 402, 39*402);
}
```

39 * 402 == 15678 ← 実行結果

図 8.8 複数個のデータを1つのprintf()で表示する

このとき、フォーマット文字列中の指定記号は、左から順に表示したいパラメータで置き換えられます。

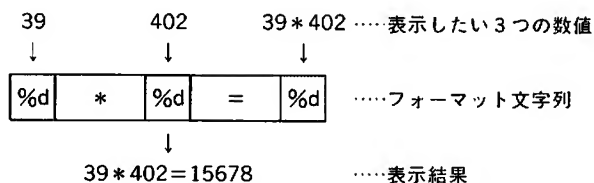
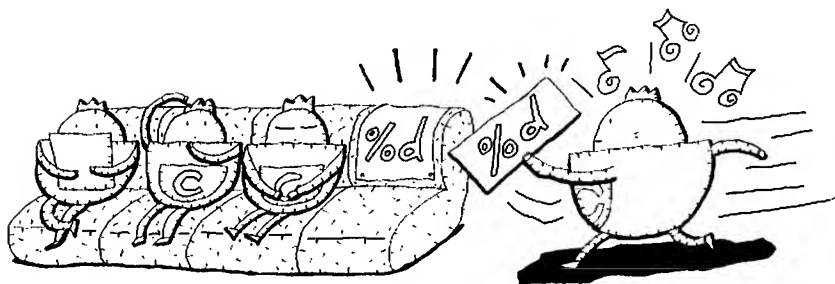


図 8.9 複数個のデータと%dの置き換えのしくみ



8.2 コントロール文字による画面制御

MSX には、コントロール文字と呼ばれる特殊な文字がいくつか用意されています。これらは画面に出力しても目に見える文字としては表示されず、その代わりに画面クリアやカーソル移動などの画面制御を行います。

■ 特別な記号を持つコントロール文字

C では、このコントロール文字も、1 つの文字として表すことができます。ただし、なにしろ目に見えない文字ですから、これをプログラムとして打ち込むときには、ちょっとした表記の工夫が必要です。

たとえば、改行させるためのニューライン文字 \textyen n はすでに紹介しましたが、この \textyen n の正体が、実は文字コード 10 に相当するコントロール文字なのです。つまり、「改行」という目に見えない存在を、キーボードから打ち込める現実の文字として表す工夫が、 \textyen n という記号だったわけです。

MSX-C では、この \textyen n を含めて、表 8.2 に示す 7 種類の記号がコントロール文字を表すために利用できます。

記号	10進文字コード(16進/8進)	機 能
\textyen a	7 (0x07/007)	ビーブ音を鳴らす
\textyen b	8 (0x08/010)	カーソルを 1 文字左に移動させる
\textyen t	9 (0x09/011)	カーソルを 8 の倍数の位置に進める
\textyen n	10 (0x0A/012)	カーソルを次の行の先頭に進める
\textyen v	11 (0x0B/013)	カーソルを画面左上に移動させる
\textyen f	12 (0x0C/014)	画面をクリアする
\textyen r	13 (0x0D/015)	カーソルを現在行の先頭に戻す

表 8.2 特別な記号で表されるコントロール文字

これらのコントロール文字を画面に出力すると、上の表に示した動作が実行されます。文字として出力されるならば、どんな方法でもかまいません。たとえば画面をクリアする¥f(文字コード12)には、次の3つの出力方法が考えられます。

```

putchar('¥f');
puts("¥f");
printf("%c", 12);

```

} どれも画面クリアを行う

■ 文字コードで表すコントロール文字

MSX で利用できるコントロール文字は、上で述べた7種類だけではなく、このほかに表 8.3 にあげたものが用意されています。ただし C では、これらについては¥n や ¥t のような記号を定めていません。

10進文字コード(16進/8進)	機 能
1 (0x1/01)	グラフィック文字の始まりを示す
27 (0x1b/033)	エスケープシーケンスの始まりを示す
28 (0x1c/034)	カーソルを1文字右に移動させる
29 (0x1d/035)	カーソルを1文字左に移動させる
30 (0x1e/036)	カーソルを1文字上に移動させる
31 (0x1f/037)	カーソルを1文字下に移動させる
127 (0x7f/0177)	カーソルを1文字左に戻し、文字を消去する

表 8.3 特別な記号が付けられていないコントロール文字

では、これらのコントロール文字はプログラム中で利用できないのかというと、そんなことはありません。実は C では ¥x の後に 16 進文字コードを続けることによって、どんな文字でも表すことができるからです。たとえばカーソルを1文字下へ移動させるコントロール文字(文字コード 0x1f)は、¥x1f と表現できます。

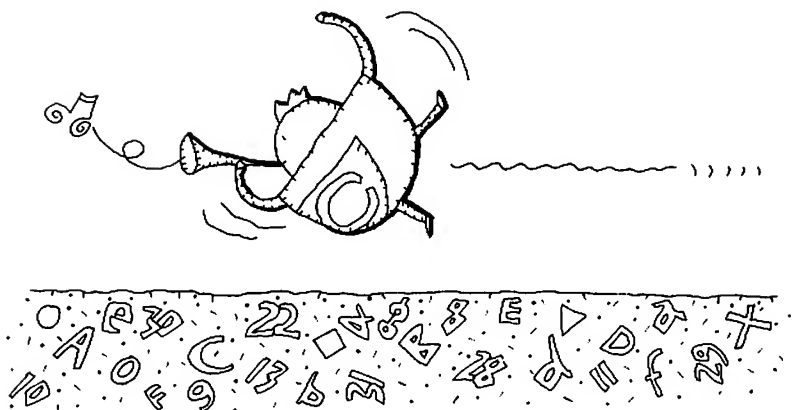
```
puts("ナ¥x1f ナ¥x1f メ");
```

..... 「ナナメ」と斜めに表示する

文字コードは16進数以外に8進数を使うことも可能です。その場合は単独の¥の後に8進数で表した文字コードを続けます。たとえば¥x1fは次のように8進コードで¥37とも表せるのです。

```
puts("ナ¥37 ナ¥37 メ");
```

..... 上と同じ結果になる



8.3 エスケープシーケンス

前節で述べたコントロール文字の中に、¥33(¥x1b)というものがありました。これは少々変わった機能を持っていて、単独で出力しても画面には何もおきません。しかし、この文字に続けて、ある特定の文字の並びを出力すると、いろいろな効果が得られます。

■ エスケープシーケンスを使ってみる

たとえば、¥33, y, 4, の3文字をこの順に出力すると、カーソルがアンダーライン「_」の形に変わってしまいます。

```
#include <stdio.h>
main()
{
    puts("¥33y4");
}
```

実行結果
カーソルの形が「_」になる

図 8.10 エスケープシーケンスの使用例

このように¥33の文字で始まる文字の並びを、一般にエスケープシーケンスと呼びます。MSXには、以下の表に示す16種類の画面制御用エスケープシーケンスが用意されています。

これらのエスケープシーケンスは、実際に使ってみないと動作を理解しにくいかもしれません。以下に続く項で、いくつか具体的な使い方を紹介することにしましょう。

エスケープシーケンス	機 能
¥33J	カーソルの位置から画面の最後までをクリア
¥33K	カーソルの位置から行の最後までをクリア
¥33L	カーソルの位置に 1 行差し込み
¥33M	カーソルの位置にある 1 行を削除する
¥33Y??	カーソルを??で指定した位置に移動
¥33x4	カーソルの形を「■」に変える
¥33x5	1 文字ごとのカーソル表示を止める
¥33y4	カーソルの形を「_」に変える
¥33y5	1 文字ごとにカーソルを表示させる
¥33A	カーソル上移動←¥36 (¥x1E) と同じ
¥33B	カーソル下移動←¥37 (¥x1F) と同じ
¥33C	カーソル右移動←¥34 (¥x1C) と同じ
¥33D	カーソル左移動←¥35 (¥x1D) と同じ
¥33E	画面クリア ←¥f と同じ
¥33H	カーソルを画面左上に移動←¥v と同じ
¥33j	画面クリア ←¥f と同じ

表 8.4 MSX の画面制御エスケープシーケンス

■ カースル位置の指定

エスケープシーケンス「¥33Y??」を使うと、BASIC の LOCATE 命令と同様にカーソルの位置を指定できます。ただし「?」の部分には、それぞれ y 座標と x 座標を指定する文字がひとつずつは入ります。

座標を指定する文字は、その文字コードが(座標+20h)に相当するものです。たとえば座標 3 を指定するのは文字コード 23h の「#」ですし、座標 6 ならば文字コード 26h の「&」ということになります。

ですから、次のように「¥33Y #&」という文字列を出力すれば、BASIC の「LOCATE 6, 3」と同じ効果が得られます(エスケープシーケンスの座標指定の順番は、y 方向が先、x 方向が後と LOCATE 命令とは逆なので注意してください)。

```
#include <stdio.h>
main()
{
    puts("¥33Y#&");
    puts("I'm here");
}
```

実行結果

座標(6,3)の位置に「I'm here」と表示

図 8.11 エスケープシーケンスによるカーソル位置の指定

エスケープシーケンスは、どんな手段を使っても正しい順番で文字を送りさえすれば、うまく働いてくれます。次のように `putchar()` で 1 文字ずつ出力してもよいのです。

```
putchar('¥33');
putchar('Y');
putchar('#');
putchar('&');
```

あるいは、`printf()` の `%c` 指定を使って、次のように書くこともできます。この方法は、いちいち文字コードの表を見る必要がなくて、いちばん便利かもしれません。

```
printf("¥33Y%c%c", 0x20 + 3, 0x20 + 6);
```

■ 行の挿入と削除

エスケープシーケンス「¥33L」は、現在カーソルが置かれている行以降を 1 行ずつ下へずらし、カーソルの位置に空行をあける働きがあります。このとき、もともと画面のいちばん下にあった行は消されてしまいます。

なお、このエスケープシーケンスを出力しても、カーソルは画面上の同じ位置、つまり挿入された空行の上にとどまります。

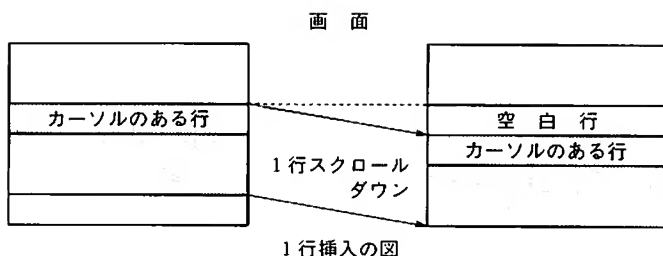


図 8.12 エスケープシーケンス「¥33L」を出力したときの動作

このシーケンスは、画面全体をスクロールダウンさせたい場合にも利用できます。それには次のように、カーソルを画面の最上行に移動させ、そこで1行挿入を行えばよいのです。

```
puts("¥v¥33L¥33L¥33L");
```

「¥33L」と正反対の動作をするのが、「¥33M」のエスケープシーケンスです。こちらは現在カーソルが置かれている行を削除し、以下に書かれていた内容を全体的に1行ずつ上へ繰り上げます。その結果、画面の最下行は空白行となります。この場合も、やはりカーソルは画面上の同じ位置にとどまります。

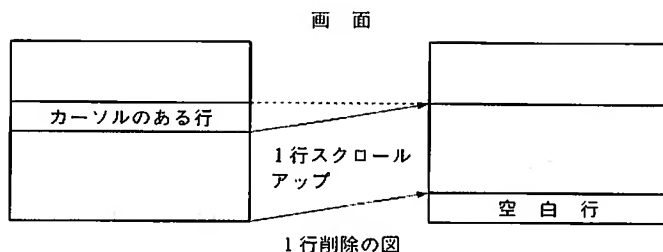


図 8.13 エスケープシーケンス「¥33M」を出力したときの動作

■ カーソルのちらつき防止

MSX-C のプログラムで文字を表示するとき、画面上でカーソルがチラチラ点滅するのが非常に気になることがあります。

たとえば次に示すプログラムを実行してみてください。これは矢印を画面の左端からスイーッと動かすプログラム、……のはずだったのですが、実際にはカーソルのちらつきが邪魔をして、とても見てられません。

```
#include <stdio.h>
main()
{
    int i;

    for (i = 1; i <= 30; ++i)
        puts(" -->%b%b%b");
}
```

図 8.14 カーソルがチラついて見づらい例

これは、MSX 本体のカーソル表示機能が通常は ON になっているため、文字をひとつ表示することにならずカーソルも一緒に表示されてしまうからです。

その邪魔なカーソルを消すエスケープシーケンスが「¥33x5」です。プログラムの先頭で、

```
puts("¥33x5");
```

というシーケンスを出力しておくと、カーソル表示機能が OFF に設定され、文字がちらつくことはなくなります。

このエスケープシーケンスは、初めに 1 回出力しておけばプログラムが終了するまで効果が続きます。また、カーソル表示機能を OFF にしておいても、キーボード入力の際にはカーソルは自動的に表示されますから、機能の ON/OFF をいちいち切り換える必要はありません。

84

そのほかの話題

画面表示に関して、ここまでの説明で述べきれなかった話題がいくつか残っています。最後にそれらをまとめて紹介しておきましょう。これでもう表示できない文字は1つもなくなるはずですよ。

■ グラフィック文字の表示

MSX には、グラフィック文字と呼ばれる特殊な文字が存在します。これは画面の上では1個の文字なのですが、表示するためには2バイトのデータを出力しなければなりません。

次の表にグラフィック文字を表示するためのグラフィックシーケンス一覧を示します。見ておわかりのとおり、かならず¥1 のコントロール文字で始まり、さらにもう1つ文字が続く形式になっています。

これらのグラフィック文字は、見かけは1文字でも実際には2バイトのデータですから、文字定数とすることはできません。たとえば、 π という文字を次のように `putchar()` で表示しようとするとコンパイルエラーになります。

```
putchar('π');    ← 誤り
```

したがって、これは次のように1バイトずつに分けて出力するか、

```
putchar('¥1');  
putchar('P');
```

あるいは `puts()` を使って次のように書く必要があります。

```
puts("¥1P");
```

シーケンス	文字(VRAMコード)	シーケンス	文字(VRAMコード)
¥1 @	(00h)	¥1P	π(10h)
¥1 A	月(01h)	¥1Q	⊥(11h)
¥1 B	火(02h)	¥1R	⌊(12h)
¥1 C	水(03h)	¥1S	⌋(13h)
¥1 D	木(04h)	¥1T	⌈(14h)
¥1 E	金(05h)	¥1U	⌉(15h)
¥1 F	土(06h)	¥1V	⌊(16h)
¥1 G	日(07h)	¥1W	⌋(17h)
¥1 H	年(08h)	¥1X	⌈(18h)
¥1 I	円(09h)	¥1Y	⌉(19h)
¥1 J	時(0Ah)	¥1Z	⌊(1Ah)
¥1 K	分(0Bh)	¥1[⌋(1Bh)
¥1 L	秒(0Ch)	¥1¥	×(1Ch)
¥1 M	百(0Dh)	¥1]	大(1Dh)
¥1 N	千(0Eh)	¥1^	中(1Eh)
¥1 O	万(0Fh)	¥1_	小(1Fh)

注：VRAMコードとは、VRAMに直接データを書き込んで表示する場合に用いるコードです。この表にあげられていない通常の文字については、文字コード=VRAMコードです。

表 8.5 グラフィック文字を表示するためのシーケンス

■ クォート記号や¥記号の表示

「¥」で始まる特殊な文字表現に次の3種類があることは、すでに見てきました。

① とくに記号の定められたコントロール文字

(¥a, ¥b, ¥t, ¥n, ¥v, ¥f, ¥r)

② 16進数による文字コード表現

(「¥x」に続く16進数)

③ 8進数による文字コード表現

(「¥」に続く8進数)

ここでさらに4番目として、次のような場合を付け加えましょう。

④¥の後ろの文字が上の①～③のどれにも相当しない場合、

¥? という表現は「?」の位置の1文字を表す

つまり¥の後ろが、a, b, t, n, v, f, tではなく、xに続く16進数でもなく、8進数でもない場合、それは¥の後ろに書かれた文字そのものを表すのです。たとえば「¥Z」は「Z」という文字を表すことになります。

もっとも、Zが¥Zと書けたからといって、何もうれしくありません。この表現が役にたつのは、プログラム上で特別な役割を持つ文字の意味を打ち消し、その文字自身をデータとして扱いたい場合です。

まず、puts()で表示する文字列の中にダブルクォートを含めたいときがあります。通常はダブルクォートが出てくると、そこで文字列は終わりとみなされてしまいますが、¥記号を付けると、ダブルクォート記号をデータとして文字列中に含められます。

また、シングルクォート自身を文字定数としたい場合があります。一般的なCでは「'''」という書き方はエラーとなるため、やはりこの場合も¥記号を付けてシングルクォート記号自身を表す必要があります。

3番目は、記号「¥」を文字データとしたい場合です。このときも¥¥と書くことで、最初の¥が次の¥の意味を打ち消して、¥という文字自身を表すことができます。

```
#include <stdio.h>
VOID main()
{
    puts("コレハ ¥レンジャウ¥ デ'ス¥n");
    putchar('¥');
    putchar('O');
    putchar('K');
    puts("おネダ'ン ハ ¥¥360 ナ");
}
```

```
コレハ "レンジャウ" デ'ス
'OK
おネダ'ン ハ ¥360 ナ
```

← 実行結果

図 8.15 クォート記号と¥記号の表示例

9章

関数は新しい世界を 開くか



前章まで紹介してきた機能は、実際のところ BASIC でも頑張れば実現可能なことで、C の絶対的なメリットではありませんでした。それでは、それぞれの優れた点であると胸を張っていえる部分はどこにあるのか。これは筆者の個人的見解ですが、やはり関数を自由に作成して使えることではないかと思います。

関数はプログラムの構造をはっきりさせ、プログラミングのあいまいさをなくし、さらに関数を蓄積していくことによってプログラム開発の手間を大きく省くことができます。C の関数には、自分で C 言語の体系を拡張していける快感があります。関数を利用できるようになると、プログラミングの世界はぐんとひろがります。

91

関数の作成法と その使い方

プログラムを作る場合、それが大きなサイズのプログラムになればなるほど、全体を一度に書き上げることが難しくなります。そのため、どのようなプログラミング言語にも、プログラムをある限られた機能ごとに分け、まず各部分を完成させた上で全体を組み合わせる手段が用意されているものです。たとえば BASIC ではサブルーチン (GOSUB 命令) が利用できました。

C では関数とその役割を担っています。といっても、数学の時間にサインコサインとかいって苦しめられた例のやつではなく、C では「ある機能を持った実行単位」のことを関数と呼びます (どちらも英語では function というのです)。

ここでは C の最も簡単な部類に属する関数から、順に高度な機能のものまで、実際にいろいろな関数の例を見ていくことにします。

■ 関数を定義する

関数を使うためには、それに先だって関数の定義を行う必要があります。関数の定義とは、その関数に実行させたい内容を指定して、さらに関数名をつける作業です。

ただしこれから述べていくように、渡されたパラメータを使うかどうか、あるいは戻り値を返すかどうかによって、関数定義の方法も少しずつ異なります。まずはもっとも簡単な、パラメータなし、戻り値なしの関数を例にとりて、基本的な関数定義の方法を紹介しましょう。

● いちばん簡単な関数の定義

いちばん簡単な C の関数は、渡されるパラメータも持たず、戻り値も返さないものです。これは実行したい文をいくつかブロックにまとめて、それに

名前をつけただけの図 9.1 のような形式で定義されます。

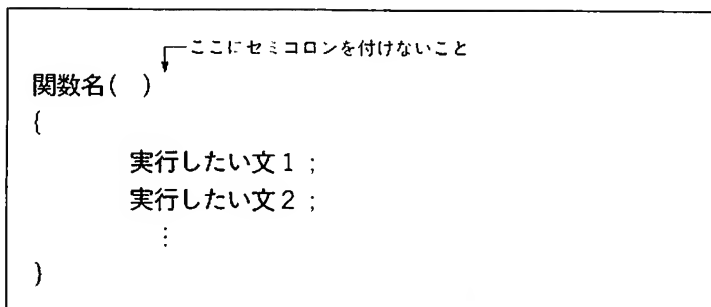


図 9.1 いちばん簡単な関数定義の形式

たとえば図 9.2 は、文字を並べて箱を描くプログラムの例ですが、ここでは横に一本線を引く `yoko()` という関数と、箱の両脇の縦線を書く `tate()` という 2 つの関数を定義して使っています。

```

#include <stdio.h>

yoko()
{
    puts("+----+");
    putchar('\n');
}

tate()
{
    puts("I      I");
    putchar('\n');
}

main()
{
    int i;

    yoko();
    for (i = 1; i <= 4; ++i)
        tate();
    yoko();
}

```

.....yoko()を実行する
.....tate()を4回繰り返す
.....yoko()を実行する

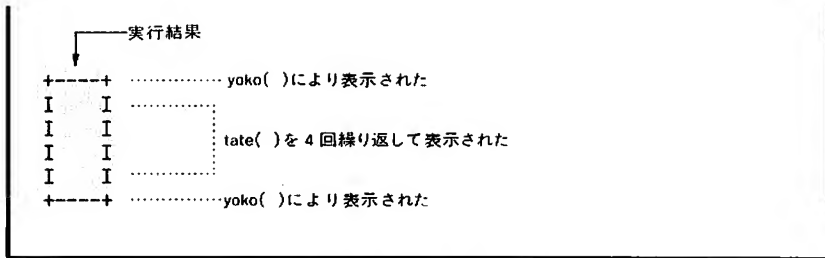


図 9.2 簡単な関数の定義とその使用例

いったん関数が定義されれば、それを実行するのはごく簡単。たとえば上のプログラムの `yoko()` という関数ならば、その名前を使って「`yoko();`」という関数呼び出し文を書くだけです。こうすると、`yoko()` の関数定義の中で指定したとおりに文が実行されるのです。

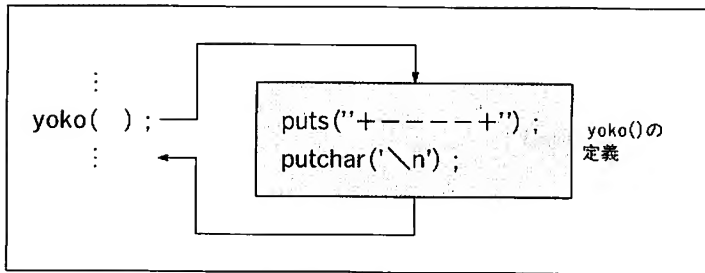


図 9.3 関数呼出と実行のしくみ

このように自分で定義した関数は、`puts()` や `putchar()` などのライブラリ関数とまったく同様に利用できます。if 文や for 文の中で使うことも自由です。図 9.2 のプログラムに次のような部分がありますが、この for 文では `tate()` という関数が 4 回繰り返して実行されるわけです。

```

for (i = 1; i <= 4; ++i)
    tate();
  
```

ここであげたような単純な関数は、ある作業をまとめて実行するだけです。から、機能的には BASIC のサブルーチンとほとんど変わりません。ただしサブルーチンが単なる行番号で呼び出されていたのに比べると、C の関数は意味のある関数名で呼び出せるぶん、プログラムが非常にわかりやすくなっています。

●関数定義はいつ行うか

こうして自分で定義した関数を利用して、さらに別の関数を定義することも可能です。ただし 4 章でも述べたように、関数を組み合わせて別の関数を作るときは、どの関数も使用する前にその存在がコンパイラにわかるようにしておかなくてはなりません。

一番簡単なのは、さきに関数定義を行い、そのあとで利用することですが、この順序をとらない場合(関数が定義される前にそれを利用する場合)は、とりあえず関数の使用宣言だけ行います。

本来この宣言は「関数の型」を考えにいれて行うべきなのですが、いま扱っているような単純な関数では、次の形式で宣言すればよいでしょう。

```
extern 関数名();
```

4 章では省略していましたが、正しくは extern を付けます。図 9.4 は、この関数宣言を使ったプログラムです。関数の宣言は 2 行目のように、すべての関数定義の外側で行ってもかまいませんし、12~13 行目のように、その関数を呼び出したい関数定義の中で独自に宣言してもかまいません。

前者の場合、関数の使用宣言はプログラム全体を通して有効となり、以後は特別な宣言なしに自由に box() という関数を使えるようになります。後者の場合、tate() と yoko() の 2 つの関数の宣言は、この宣言を行っている box() の中だけで有効となり、ほかの関数が tate() や yoko() を利用したければ、それらが自分自身でまた宣言を行う必要があります。

```

0: #include <stdio.h>
1:
2: extern box();
3:
4: main()
5: {
6:     box();
7:     box();
8: }
9:
10: box()
11: {
12:     extern tate(); ←
13:     extern yoko(); ←
14:
15:     yoko();
16:     tate();
17:     tate();
18:     yoko();
19: }
20:
21: yoko()
22: {
23:     puts("+----+");
24:     putchar('\n');
25: }
26:
27: tate()
28: {
29:     puts("I    I");
30:     putchar('\n');
31: }

```

定義の中で宣言してもよい
(この場合、tate()とyoko()の関数宣言は、
box()の中だけで有効となる)

図 9.4 関数宣言を行って、未定義の関数を利用する方法

なお、この本では面倒な手間を省くため、できるだけ関数宣言は行わずに関数定義の並べ方を工夫して対処していくことにします。

●関数は自分専用の変数を持てる

それぞれの関数は、その中で自分専用の変数を使うことができます。この変数は図 9.5 に示したように、関数本体のブロックの先頭(実行文が出て来る前)の位置で宣言します。

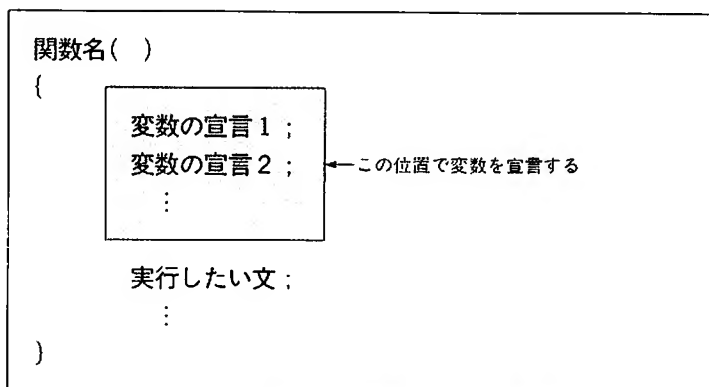


図 9.5 関数の中で変数を使いたい場合

図 9.6 は、アスタリスク「*」を 7 文字×5 行の 4 角形に並べるプログラムです。ここではまずアスタリスクを 7 文字表示して改行する `astro7()` という関数を作り、それを 5 回繰り返して呼び出しています。

```

#include <stdio.h>

astro7()
{
    int i; .....この変数iはmain( )の変数iとは無関係

    for (i = 1; i <= 7; ++i)
        putchar('*');
    putchar('\n');
}

main()
{
    int i; .....この変数iはastro7( )の変数iとは無関係

    for (i = 1; i <= 5; ++i)
        astro7();
}

*****
*****
***** ← 実行結果
*****
*****

```

図 9.6 それぞれの関数で使う変数は互いに無関係である

上のプログラムで、特に注意していただきたいのは、`astro7()`の関数定義の中でも*i*がループ変数として使われ、その`astro7()`を呼び出す`main()`の関数定義の中でも、やはり*i*がループ変数に使われていることです。

これがBASICならば、`astro7()`の実行中に*i*が書き換えられてしまうため、`main()`のループは意図したとおりには動いてくれないでしょう。しかし、Cではそうはなりません。なぜなら関数定義の中で宣言された変数は、その関数専用のものとみなされ、互いにまったく独立して扱うことができるからです。

つまり図9.6のプログラムでは、`astro7()`の中でいくら変数*i*を書き換えても`main()`の変数*i*には影響がありませんし、その逆に`main()`の中で*i*の値を書き換えても`astro7()`の変数*i*は少しも変更を受けないわけです。このように、ある関数の中だけで通用し、ほかとはまったく独立している変数を、ローカル(局所的)な変数と呼びます。

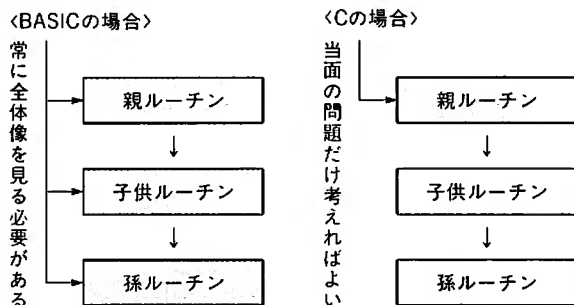


図 9.7 ローカル変数の利点

このローカルな変数は、`int` 型、`char` 型など、どんなデータ型でも使えますし、配列変数でもかまいません。

Cでは、このローカル変数のおかげで、大規模なプログラムの開発がたいへん簡単にできるようになりました。BASICの場合は、どのサブルーチンでどんな変数が使われているかということをすべて心得ておかないと、予期しない変数の書き換えていつプログラムが破綻するか知れません。

これに対してCでは、関数の中でどのような名前の変数が使われているよう

と、それを利用するほうはいっさい気にする必要がありません。関数というものは、その動作さえわかれば OK。astro7() はアスタリスク記号を 7 個表示する、ただそれだけでよいのです。

●パラメータを持つ関数の作成

さて、図 9.6 のプログラムで作った astro7() は、必ず 7 個のアスタリスクを表示する固定動作の関数でしたが、C では与えるパラメータによって、いろいろと動作を変えられるような関数も簡単に定義できます。

パラメータを持つ関数の定義形式は次のとおりです。

```
関数名(パラメータ変数名 1 ; パラメータ変数名 2 .....)  
パラメータ変数の型宣言 1 ;  
パラメータ変数の型宣言 2 ;  
    :  
{  
    ローカル変数の宣言  
    :  
    実行したい文  
    :  
}
```

図 9.8 パラメータを持つ関数の定義の形式

パラメータの個数はいくつでも、どのデータ型でもかまいませんが、基本的に 1 つのパラメータが受けとれるデータは 1 つです。配列のようにまとまったデータをごそっと渡すことはできません(ただしこれには抜け道があり、10 章で説明するポインタを利用すると、配列を受け取る関数を作ることが可能になります)。

図 9.9 に、astros() という関数の定義とその使用例を示しました。この astros() という関数は、カッコの中に適当な数値データを入れて、

```
astros(数値);
```


という形式で用いると、指定した個数ぶんのアスタリスクを表示します。この数値は 16 行目のように定数でも、19 行目のように変数を使っても、あるいは 22 行目のように計算式で指定してもかまいません。

```

0: #include <stdio.h>
1:
2: astros(n) .....astros( )がnというパラメータを持つことの宣言
3: int n; .....nがint型であることの宣言
4: {
5:     int i;
6:
7:     for (i = 1; i <= n; ++i) .....1からnまで繰り返す
8:         putchar('*');
9:     putchar('\n');
10: }
11:
12: main()
13: {
14:     int i;
15:
16:     astros(5); .....パラメータに5を指定する
17:
18:     for (i = 1; i <= 4; ++i)
19:         astros(i); .....変数でも指定できる
20:
21:     for (i = 1; i <= 3; ++i)
22:         astros(4-i); .....計算式でも指定できる
23: }

```

実行結果

```

***** .....パラメータ5を与えてastros( )を実行した結果
*
** .....1, 2, 3, 4とパラメータを変えながら
*** .....astros( )を実行した結果
****
* .....3, 2, 1とパラメータを変えながら
** .....astros( )を実行した結果
*

```

図 9.9 表示するアスタリスクの数を変えられる関数

●パラメータ変数に関する注意

図 9.10 は、第 1 パラメータの文字 `c`(char 型)を、第 2 パラメータ `n`(int 型)の個数だけ表示する `putline()` という関数の定義です(これは 4 章の“TRIANGLE.C”というプログラムの中に出てきたものです)。

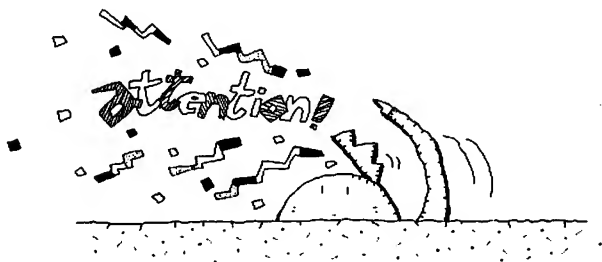
このように、パラメータを受け取る変数(以後パラメータ変数と呼びます)は、それ以外の通常の変数と同様に、関数の定義の中で自由に値を参照したり変更することが可能です。

```
putline(ch, n)
char ch;
int n;
{
    int i;

    for (i = 1; i <= n; ++i)
        putchar(ch);
}
```

図 9.10 `putline()` の定義

関数を定義するときのパラメータ変数の型と、それを実際に利用するときと与えるデータの型は、かならず等しくなるように注意してください。特に MSX-C では関数の第 1 パラメータについては、char 型とそれ以外のデータ型の区別が厳しく、これを間違えるとプログラムが思いどおりに動作してくれないこともあります。



たとえば上の `putline()` は、第1パラメータが `char` 型であることを前提に作られていますから、ここに `char` 型以外のデータを使ってはいけません。文字 `A` を5個表示させるのに、

```
putline('A', 5);           ..... 文字定数は char 型
putline((char)65, 5);      ..... キャストで char 型に合わせる
putline((char)0x41, 5);    ..... //
```

などを書くことはできますが、

```
putline(65, 5);           ..... 65 は int 型
putline(0x41, 5);         ..... 0x41 は unsigned 型
```

と書いてしまうと正しく動作しないのです。

気をつけなくてはならないのは、パラメータを計算式で指定する場合のデータ型です。たとえば、次のような呼び出しを考えてください。

```
putline('A'+i, 10);       ..... 変数 i は int 型とします
```

この場合、「`A'+i`」という式は `char` 型+`int` 型ですから、7章で説明したとおり、`int` 型になってしまいます。この式は `char` 型にキャストして、次のように書かなくてはなりません。

```
putline((char)('A'+i), 10);
```

●関数はプログラムをわかりやすくする

8章でカーソルを指定位置に移動させるエスケープシーケンスを紹介しました。これは、次のような文字コードを持つ4文字を順に出力することにより、カーソル移動を実現するものでした。

```
¥033, Y, Y 座標+32, X 座標+32
```

そこで、`x` と `y` の座標をパラメータに与えるとこのエスケープシーケンスを出力するような関数を作ってみます。関数の名前は BASIC の `LOCATE` 命令にならって `locate()` としましょう。図 9.11 に、`locate()` 関数の定義とその使用例を示します。

```

#include <stdio.h>

locate(x, y)
int x, y;
{
    printf("%03Y%c%c", y+32, x+32);
}

main()
{
    int i;

    for (i = 0; i <= 20; ++i) {
        locate(i, i);
        printf("[%d,%d]", i, i);
    }
}

```

図 9.11 カーソル移動関数 locate() の定義と利用

さて、このプログラムを見てわかるとおり、locate() の定義の本体は、たった 1 行だけです。ですから、これを関数化したからといって、プログラムの手間を省くという点では、まったく役に立っていません。実際次のように書くほうがプログラムのサイズもずっと小さくなります。

```

#include <stdio.h>
main()
{
    int i;

    for (i = 0; i <= 20; ++i) {
        printf("%03Y%c%c", i, i); .....1行で書けるが意味がわかりにくい
        printf("[%d,%d]", i, i);
    }
}

```

図 9.12 locate() を定義しないで書いたプログラム

しかし、図 9.11 のように、あえて locate() という関数を作ることには、プログラムの手間を省くという以上の大きなメリットがあるのです。それはプ

プログラムのわかりやすさです。つまり、

```
printf("¥033Y%c%c", i, i);
```

と書いても、これがカーソル移動を意味するとは、なかなか理解できません。それに対して、

```
locate(i, i);
```

と書かれていれば、誰が見てもこれはカーソルを (i, i) の位置に移動させるものだと判断できます。大規模なプログラムを作るときには、このようなプログラムの理解しやすさという要素も非常に大事です。その意味からも関数はどんどん利用すべきなのです。

●関数の実行を途中で終わらせる

関数は、通常はその定義の最後の文まで実行すると終わりになりますが、それ以外に次の return 文で強制的に終了させることもできます。

```
return;
```

この return 文を使った例として、さきほどの locate() を、座標が (0,0) ~ (39,19) の範囲外であればカーソル移動を行わずに即座に終了させるように改良してみましょう。この定義は図 9.13 のようになります。

この return 文は、いつでもどこでも実行可能です。ループが何重にも入れ子になっているような状態の奥底からでも、いっきに関数を終了させることができます。

```
locate(x, y)
int x, y;
{
    if (x < 0 || x > 39 || y < 0 || y > 19) ..... 画面からハミ出した時は...
        return; ..... 戻る
    printf("¥033Y%c%c", y+32, x+32);
}
```

図 9.13 return 文は関数を中断できる

■ 戻り値を返す関数

5 章でも少し触れましたが、C の関数には、なんらかのアクションを目的として使う `putchar()` のようなものと、戻り値を利用するために使う `getch()` のようなものの 2 種類があります。

前者については、すでに前の節で述べました。ここでは戻り値を返す関数の宣言とその使用法を説明します。

● 戻り値を持つ関数の宣言

戻り値を持つ関数を宣言するには、図 9.14 のように関数名の前に戻り値のデータ型を指定し、戻り値指定付きの `return` 文で戻り値を返します。

```

戻り値の型 関数名( )
{
    実行したい文
    ⋮
    return 戻り値;
}

```

図 9.14 戻り値を持つ関数の定義

戻り値を持つ関数の例を図 9.15 に示します。ここで定義している `add()` は渡された 2 つのパラメータの和を返す関数です。

このような関数の戻り値は、変数に代入したり、計算式の中の 1 つのデータとして利用することができます。たとえば、

```
i = add(1, 2);      …… i に (1+2) が代入される
```

と書けば、`add()` は `1+2` を計算して 3 という戻り値を返し、それが変数 `i` に代入されます。次のように、関数に渡すパラメータの中でさらに関数呼び出しを書くこともできます。

`i = add(1, add(2, 3));` …… `i`に`(1+(2+3))`が代入される

```
#include <stdio.h>

int one()
{
    return 1; ……………つねに1を返す
}

int add(x, y)
int x, y;
{
    return (x + y); ……………2つの数値の和を返す
}

main()
{
    printf("1 + 1 == %d\n", add(1, one()));
}
```

図 9.15 戻り値を持つ関数の例

● 戻り値のデータ型

関数の戻り値のデータ型は、その宣言で指定した型に一致します。これは `int` 型でも `char` 型でも `unsigned` 型でもかまいませんが、戻り値として返せるデータはたった1つです(ただし10章で説明するポインタを利用すると、複数の戻り値を返す関数を作ることも可能になります)。

`return` 文で返すデータの型は、自動的に指定した関数のデータ型に合わされます。たとえば図9.16を見てください。これは、与えられたパラメータに `n` を加えたコードの文字を返す関数 `next()` の定義です。

```
char next(c, n)
char c;
int n;
{
    return (c + n);
}
```

図 9.16 戻り値のデータ型は関数宣言が決める

この関数では、return 文に指定された戻り値は char 型+int 型で、int 型のデータになってしまうように思えます。しかしこの関数は char 型を返すと宣言されているため、結果は自動的に char 型に変換されるのです。つまり、この next() という関数の戻り値は、かならず char 型であることが保障されています。たとえば次のように、putchar() のパラメータに渡しても問題はありません。

```
putchar(next('A')) ;    …… 文字 B が表示される
```

● VOID 型について

かつては戻り値を持たない関数は「型がない関数」として扱われていましたが、最近では、戻り値を持たない関数を void 型という特殊な型名で呼び、形式上どんな関数にも戻り値のデータ型がある、とすることがあります。ただし MSX-C では、これを大文字の VOID 型と呼んでいます。

VOID 型の指定は、ほかのデータ型と同様に関数名の前に置きます。たとえば、図 9.17 は画面をクリアする cls() という関数の定義です。

```
VOID cls()
{
    putchar('%f');
}
```

図 9.17 戻り値を返さない関数は VOID 型と宣言してもよい

戻り値を持たない関数に対して VOID の宣言を行っても行わなくても、関数の動作には変わりありません。しかし、これによって戻り値を返さない関数であることが強調され、プログラムの意味がわかりやすくなるという効果があります。MSX-C のサンプルプログラムなどを見ても、戻り値を持たない関数は、かならずこの VOID 型が宣言されています。

92

記憶クラスとスコープ

関数のなかでローカルな変数が利用できることはすでに説明しましたが、ときには複数の関数間で同じ変数を共有したい場合もあります。このようなときは、変数が関数ごとに独立しては困ります。またローカルな変数といっても、一時的に作業用として使えればよい場合もあれば、関数を呼び出すごとに前回使ったデータをまた参照したい場合もあります。

ここでは、これらの要求に応える変数の宣言の方法と、その具体的な使用方法について説明しましょう。

■ 変数の記憶クラス

Cの変数は、それぞれ記憶クラスと呼ばれる性質を持っています。記憶クラスは変数の寿命を決定します。変数の寿命とは、関数を呼び出すごとに新しい変数が用意されるのか、あるいはプログラムの実行開始時から終了時まで常に同じ変数が参照されるのかどうかを意味します。

MSX-Cの変数には、寿命の違いによって、表 9.1 に示す 2 つの種類の記憶クラスがあります。

記憶クラス	変数の寿命
auto変数	関数を呼び出すごとに新しく用意される
static変数	プログラム全体を通して同じ変数が使われる

表 9.1 変数の記憶クラス

● その場かぎりの auto 変数

ここまで使ってきた変数は、すべて auto 変数と呼ばれる記憶クラスに属するものでした。この auto 変数は、関数が呼び出されるごとにあるメモリ領

域に一時的に用意され、その関数の実行が終了すると捨てられてしまいます。

関数内で宣言した変数は、黙っていれば自動的に auto 変数とみなされますが、auto 変数であるということを強調したいときには、次のような宣言を行うこともできます。

```
auto int i;
```

もっとも、この auto は、あってもなくてもプログラムの動作はまったく変わらないため、いちいち宣言されることはまずありません。普通は省略してしまいます。

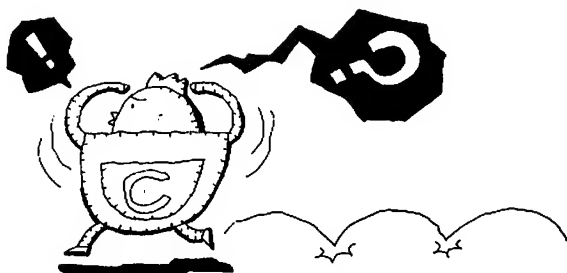
●いつまでも覚えている static 変数

通常のプログラムで使う変数は、auto 変数であっても何も問題はないのですが、用途によっては、関数が終了してもその中で使った変数の値を消したくない場合があります。

このような場合には、static 変数を使用します。static 変数は、次のように static というキーワードを頭に付けて宣言します。

```
static int i;
```

図 9.18 のプログラムは、この static 変数を利用して作った乱数関数 rnd() の例です。rnd() の中には r という static 変数が用意されていて、呼び出されるごとに「 $r = r * 257 + 1$;」という計算が行われます。このように前回の値を覚えておいてくれるのは、r が static 変数であるからです。



```

#include <stdio.h>

int rnd()
{
    static int r;

    r = r * 257 + 1;
    return r;
}

main()
{
    int i;

    for (i = 1; i <= 1000; ++i)
        printf("%10d", rnd());
}

```

図 9.18 static 変数の使用例

なお、運がよければ、auto 変数でも次の関数呼び出しまで値が捨てられずに残っていることがあります。それは本当に運がよかっただけ。プログラムはなによりも確実さが命です。幸運に頼ってはいけません。

■ 変数のスコープ

ここまで見てきた C の変数は、すべて 1 つの関数の中だけでしか参照できないローカルな変数でした。これに対して、BASIC で使う変数のように、サブルーチンの中でもメインルーチンの中でも自由に参照できる変数をグローバルな変数と呼びます。

このように、ある変数が通用する範囲を変数のスコープといいます。C の変数には表 9.2 に示した 2 種類のスコープがあります。

スコープ	変数の宣言方法	スコープの範囲
ローカル変数	関数の中で宣言する	1 つの関数
グローバル変数	関数の外で宣言する	1 つのプログラム

表 9.2 変数のスコープ

●関数の中だけで通用するローカル変数

ここまで述べてきたとおり、関数の中で宣言した変数は、その関数の中だけで通用するローカル変数となります。

●どこでも通用するグローバル変数

ローカル変数に対して、図 9.19 のようにすべての関数定義の外側で宣言した変数はグローバル変数と呼ばれ、プログラム全体から参照できる変数となります。ただしグローバル変数といえども、宣言の前に使用することはできません(そのため、グローバル変数はたいていプログラムの先頭でまとめて宣言されます)。

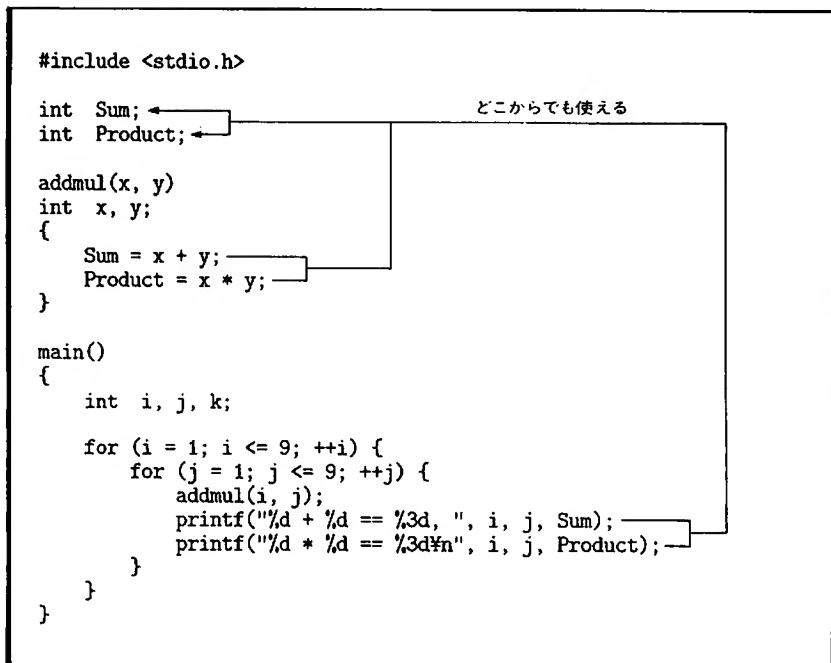


図 9.19 グローバル変数を使ったプログラム

グローバル変数を使うと、関数の間で複数のデータをやりとりすることが可能です。ここで定義している `addmul()` という関数は、2つの数値 `x`, `y` の和と積を求めるものですが、この答を両方とも関数の戻り値として返すことはできません。関数の戻り値として返せるデータはたった1つです。

そこで、ここは2つの答をそれぞれ `Sum` と `Product` というグローバル変数に代入することにします。`addmul()` を呼び出した `main()` のほうでは、関数の戻り値として答を受け取るのではなく、グローバル変数の `Sum` と `Product` を見て答を知ればよいわけです。

このように、グローバル変数は使い方によってはたいへん便利なのですが、BASICの変数と同様に、どこで値が書き換えられるかわからない欠点も持っています。グローバル変数をやたらに書き換えると、プログラムの動作が非常に理解しにくくなってしまいます。

そこで、この本では、グローバル変数はちょっと特別な変数なのだぞ、ということを強調するために、グローバル変数の名前は先頭の文字をアルファベットの大文字にするという規則を設けることにしました。

10章

ポインタと配列と文字列



マシン語でプログラミングした経験のある方はご存じだと思いますが、マシン語では、どのアドレスにどういう形でデータを記録するかをプログラマが自分で管理しなければなりません。しかしこのアドレスというやつは根が単なる数字ですから、いちいち覚えておくのも面倒ですし、間違えずに使いこなすのがなかなか大変です。

そこでCでは、アドレスをポインタという特別なデータと考えて、その扱い方をきちんと定めることにしました。おかげでCではアドレス操作が非常に簡単で便利なものとなっています。またポインタは、配列や文字列とも分かちがたい関係をもっています。

本章では、このポインタの利用法を紹介します。

10

1

ポインタを使った
プログラム

ポインタとは「あるものを指し示す存在」です。たとえばポインタという種類の狩猟犬は、獲物のいる場所を前足で指し示し、『あそこですよ御主人さま』と教えてくれるのだそうです(真偽はサダカでない)。Cでは「データを指し示すもの」をポインタと呼びます。単純なアイデアですが、これがあるおかげで、Cは大変に柔軟性を持つプログラミング言語となりました。

■ アドレス演算子「&」と間接演算子「*」

コンピュータの中では、データはメモリと呼ばれる記憶場所に蓄えられます。MSXの場合、メモリは65536個の小さな部屋に分割され、それぞれ1バイトのデータを記憶させることができます。そして、その中から特定の場所を指定するために、メモリには0番地～65535番地(0x0000～0xffff)の一連のアドレスが順に振られています。

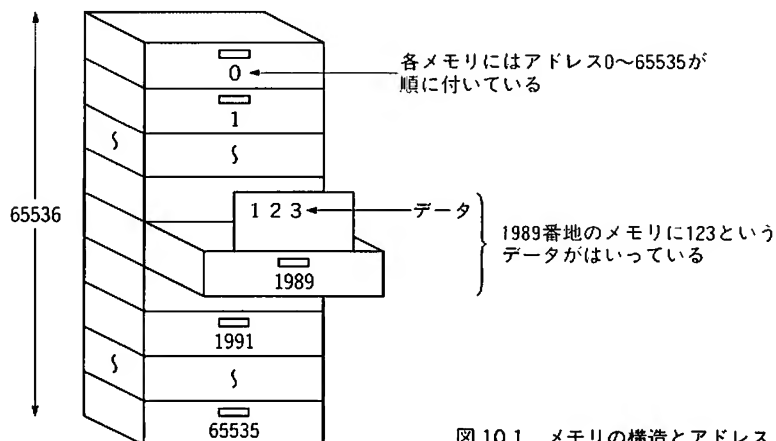


図 10.1 メモリの構造とアドレス

プログラムの中で使う変数や配列は、すべてこのメモリ上に割り当てられています。BASIC の場合は、かなり特殊なプログラムでもない限り、この変数のアドレスを知る必要はありませんでした。それに対して C では、これから述べるように、変数のアドレスを利用するといろいろ面白いプログラムを作ることが可能です。

●変数のアドレスを調べる

変数のアドレスを知るためには、& 記号で表されるアドレス演算子というものを使います。この&記号を単独で変数の前に付けると、その変数のメモリ上のアドレスが求められます。たとえば、

&i

は、変数 i のアドレスを表します。また&演算子は普通の変数だけではなく、配列変数の要素に対して用いることもできますから、

&a [i]

と書けば、配列要素 a [i] のアドレスが得られます。これらを実際にプログラムの中で使った例を図 10.2 に示しましょう。

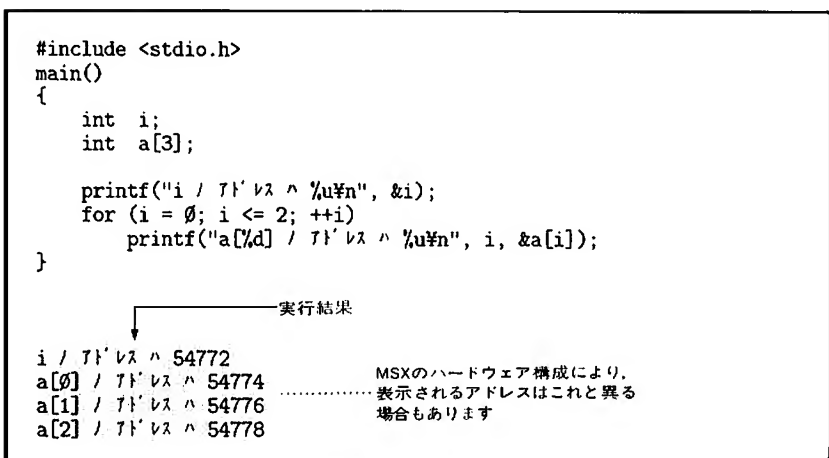


図 10.2 & 演算子で変数のアドレスを求める

●変数のアドレスに数値を加えるとどうなるか

このように&演算子を使って求められる変数のアドレスは、通常の数値データとは、ちょっと違う性質を持っています。試しに図 10.3 のプログラムを実行してみてください。ここでは int 型変数 i のアドレスと、それに 1 を加えた値、そして 2 を加えた値を表示させています。

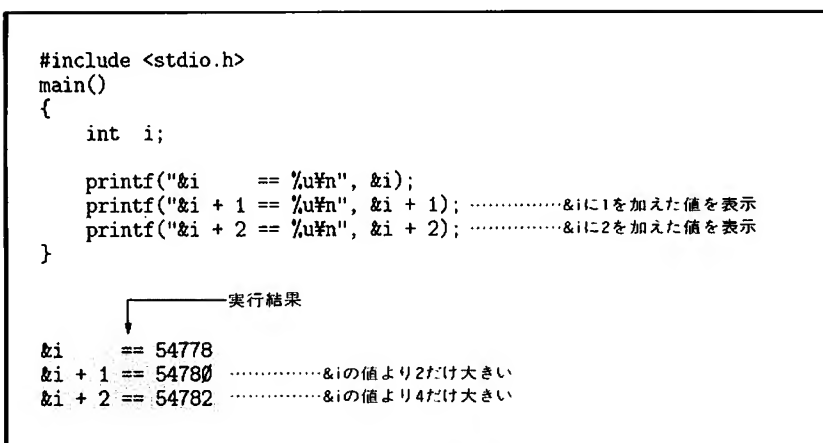


図 10.3 アドレスに数値を加える

すると図 10.3 に示すとおり、&i に 1 を加えるとアドレスは 2 増加し、&i に 2 を加えるとアドレスは 4 だけ大きくなりました。つまり、加えようとした数値の 2 倍の数がアドレスに足されているのです。

●ポインタ型のデータとは

図 10.3 のような結果が得られるのは、コンパイラが &i をポインタ型のデータとみなすからです。

ポインタ型は、アドレスを専門に扱うために用意されたデータ型です。これは通常の数値データと違い、掛け算や割り算の対象にはできません。できるのは、次のように、そのアドレスを基準としていくつか先のアドレスを求めること(ポインタの足し算)、あるいはいくつか前のアドレスを求めること(ポインタの引き算)だけです。

$\&i + n$ ($\&i$ の n 個先の変数のアドレスを求める)

$\&i - n$ ($\&i$ の n 個前の変数のアドレスを求める)

ところで、 n 個だけ先や前にある変数といっても、どのようなデータ型の変数を考えるかで結果は異なってきます。これは変数のデータ型に応じて、それがメモリ上で占めるサイズが異なるためです。

たとえば、MSX-C では `int` 型の変数は 2 バイトのメモリを使い、`char` 型の変数は 1 バイトのメモリを使います。ですから図 10.4 のように、1 つ先の変数といっても `int` 型の場合なら 2 バイト先、`char` 型ならば 1 バイト先ということになります。さきほどの図 10.3 のプログラムで $\&i$ に 1 を加えるとアドレスが 2 つ増えたのも、 $\&i$ が `int` 型の変数のアドレスだったからなのです。

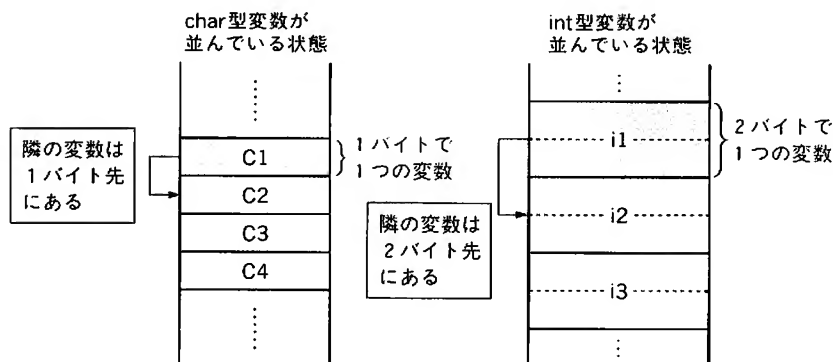


図 10.4 「1 つ先の変数」はどこにあるか

そのためポインタを扱うときは、それが `int` 型変数のアドレスなのか、はたまた `char` 型変数のアドレスなのか、つねにそのアドレスに位置する変数の型まで区別して考えなければなりません。

`int` 型変数のアドレスの性質を持つポインタを、C では `int` 型へのポインタと呼びます。同様に、`char` 型へのポインタといえば、これは `char` 型変数のアドレスに相当します。

●ポインタを変数に代入して使う

ポインタ型のデータは、同じポインタ型の変数に代入することができます。ポインタ型の変数は、次のように変数名の前にアスタリスク記号「*」を置いて宣言します。

データ型名 *変数名 ;

たとえば int 型へのポインタを格納するために p という名前の変数を使おうと思ったら、

```
int    *p ;
```

という宣言を書くことになります。

このようなポインタ型の変数も、やはりポインタ型データの1つですから、

```
p = p + 1 ;
```

という足し算は、p に 1 を加えるのではなく、「p を 1 つ先のアドレスに進める」という意味になります。つまり、この代入文によって p の指し示すアドレスは 2 だけ増加するわけです。これは、

```
++p ;
```

と書いても同様です。ポインタに対する加減算はすべて、その指し示すデータのサイズ単位でポインタを前後にいくつか移動させることになるのです。

●ポインタの指し示すデータを参照する

ポインタ型データの頭に単独の「*」を付けたものは、そのポインタの指し示すメモリの内容を表します。たとえば、

```
*p
```

というのは「p が指し示すアドレスにあるメモリの内容」を意味します。ここでは p が int 型へのポインタとして宣言されていますから、*p は int 型のデータとして扱われます。

この *p は、次のように普通の int 型の変数を使うのと同まったく同じ感覚

で、値の代入や参照ができます。

```
*p = 12345;    ..... *p への代入
i = *p;        ..... *p の参照
```

このときのメモリの状態を図で示すと、図 10.5 のようになります。つまり *p の値は、「『変数 p の内容』で示されるメモリの内容」という、2 段階の手間を通して間接的に参照されているのです。このことから、*演算子を間接演算子とも呼びます。

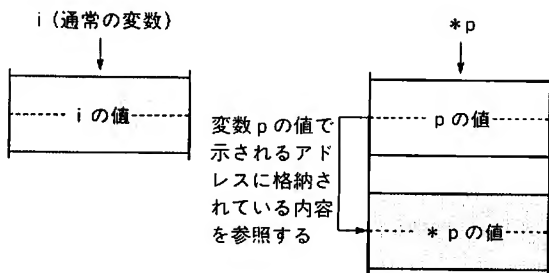


図 10.5 *p が参照される仕組み

配列とポインタの関係

前項でポインタの基本的な扱い方を紹介しましたが、これだけの説明では、いったいポインタのどこが便利なのか、なぜポインタを使う必要があるのか、いまひとつ理解できなかったと思います。

実はポインタというのは、C のほかの機能と組み合わせて使うことによって初めて大きな利用価値が生まれるのです。ここではまず、配列とポインタを組み合わせた場合を見てみましょう。

●ポインタを使って配列を扱う

ポインタに 1 を足したものは、それが int 型へのポインタであっても char 型へのポインタであっても、つねにその 1 つ先の変数を指すということは、

すでに述べたとおりです。

これが普通の変数ならば、その変数自身と隣にある変数の間にはなんの関係もありませんから、「1つ先の変数」の内容などを調べる意味はありません。

しかし配列を扱う場合になると、話が違ってきます。配列とは、そもそも int 型なら int 型のデータを順序よく並べたものですから、 $a[i]$ の 1 つ先のアドレスには、かならず $a[i+1]$ が位置しています。そのためポインタを使うと、配列をうまく扱うことが可能となります。

たとえば図 10.6 は配列 $a[3]$ の内容をすべて表示する簡単なプログラムですが、ここではポインタ変数 p の値を 1 つずつ先に進めながら、配列要素を順に表示しています。

```
#include <stdio.h>
main()
{
    int a[3];
    int *p;
    int i;

    a[0] = 1957;
    a[1] = 7;
    a[2] = 21; } 配列の中身を準備する

    p = &a[0]; .....pに配列の先頭アドレスを代入
    for (i = 0; i <= 2; ++i) {
        printf("%d " *p); .....pの指すデータを表示する
        ++p; .....pを1つ先の要素に進める
    }
}
```

1957 7 21 ←——実行結果

図 10.6 ポインタを使って配列を参照する

この図 10.6 では、ループ変数 i を使って for ループを実現しましたが、あるいはもっと直接に、 p 自身をループ変数としてしまうことも可能です。その場合は次のような for 文になります。

```
for (p = &a [0]; p <= &a [2]; ++p)
    printf("%d", *p);
```

はじめのうちは、ポインタを使ったプログラムはわかりにくいと思うのですが、ポインタの扱いに慣れてくると、かえってこのように書いたほうがプログラムがすっきりして見えてくるから不思議です。

●配列名の正体はポインタである

図 10.6 のプログラムでは、配列 `a []` の先頭へのポインタを、

```
&a [0]
```

という形で求めていましたが、これは次のように書くこともできます。

```
a
```

`a` というのは、ただ配列名をそのまま書いただけじゃないかって？ そうなのです。実は C では、配列名はその配列の先頭要素へのポインタでもあるのです。ためしに図 10.7 のプログラムを実行してみてください。&`a` [0] と `a` では、まったく同じ値が表示されるはずです。

```
#include <stdio.h>
main()
{
    int a[3];

    printf("&a[0] == %u\n", &a[0]);
    printf("a     == %u\n", a);
}
```

図 10.7 &`a` [0] と `a` は同じ値になる

また、配列名 `a` が配列の先頭要素 `a [0]` へのポインタなので、`a+1` はその次の要素の `a [1]` を指すポインタ、つまり `&a [1]` に等しいわけです。同様に考えると、`a+i` という計算式は、`&a [i]` とまったく同じアドレスを指

すポインタであることもわかるでしょう。このことは、次に述べる関数とポインタの組み合わせで威力を発揮します。

関数定義にポインタを利用する

このポインタを関数定義と組み合わせると、配列をパラメータとして渡したり、関数から複数個のデータを受け取るなど、いままでの説明にはなかった機能を持つ関数可以实现できます。

●ポインタを関数に渡すには

まず基本的な事実は、ポインタは関数のパラメータとして使うことができるといことです。たとえば図 10.8 に示すのは、int 型へのポインタをパラメータとする関数 `clear()` の定義例です。この関数は、与えられたポインタの指すメモリに 0 を書き込みます。

```
#include <stdio.h>

clear(p) .....clear( )はpをパラメータとする
int    *p; .....pはint型へのポインタである
{
    *p = 0; .....pの指すアドレスをゼロクリアする
}

main()
{
    int i;

    clear(&i); .....iをゼロクリアする
    printf("i == %d", i);
}
```

図 10.8 ポインタをパラメータとする関数

このとき、`main()`の中で、

```
clear(&i);
```

を実行すると、図 10.9 のような操作が行われて、変数 *i* に 0 が代入されます。

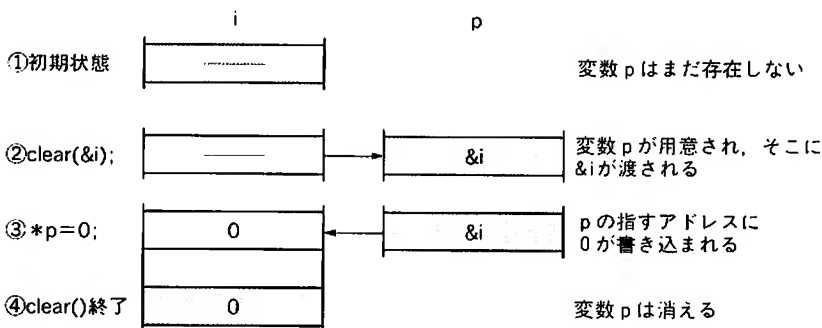


図 10.9 clear()の動作の仕組み

ここで注目して欲しいのは、clear()の実行によって、main()の使っているローカル変数 *i* が 0 になることです。

関数の中で宣言したローカル変数は、その関数の中だけでしか参照できないというのがタテマですから、通常は関数がどんな操作を行おうとも、呼び出し側の変数は変更を受けません。しかし、このようにポインタを渡すことによって、呼び出し側の変数に影響を及ぼす関数を作れるのです。

●関数から複数の値を受け取る方法

上で述べた「ポインタ渡し」のテクニックを使うと、1つの関数で、1度に複数の答を求めることができます。図 10.10 はその実例で、2つの数値の和と差を計算する addsub() という関数の定義です。

```
addsub(x, y, s, d)
int  x, y; .....xとyはint型
int  *s, *d; .....sとdはint型へのポインタ
{
    *s = x + y; .....sの指すアドレスにxとyの和を書き込む
    *d = x - y; .....dの指すアドレスにxとyの差を書き込む
}
```

図 10.10 1 度に複数の答を計算する関数

この `addsub()` を使うときは、初めの2つのパラメータに計算したい数値データ、後の2つのパラメータに答を受け取る変数へのポインタを渡します。たとえば、1000 と 700 という2数の和と差を、`sum` および `dif` という変数に代入するには、

```
addsub(1000, 700, &sum, &dif);
```

という形でこの関数を実行します。関数の実行後に `sum` と `dif` の内容を調べてみれば、それぞれ 1700 と 300 が代入されているはずです。

●配列を扱う関数の作り方

またポインタを利用すると、配列を関数のパラメータとして渡すことができるようになります。まず図 10.11 を見てください。これは、渡されたポインタ以降に存在する `n` 個ぶんの `int` 型データを表示するための、`nprint()` という関数の定義です。

```
nprint(p, n)
int *p;
int n;
{
    while (--n >= 0) { .....ループをn回繰り返す
        printf("%d ", *p); ..... *pの値を表示し、
        ++p; ..... pを1つ先に進める
    }
}
```

図 10.11 配列の内容を表示する関数

この `nprint()` を使って、たとえば5個の要素を持つ配列 `a[]` の内容をすべて表示させるには、

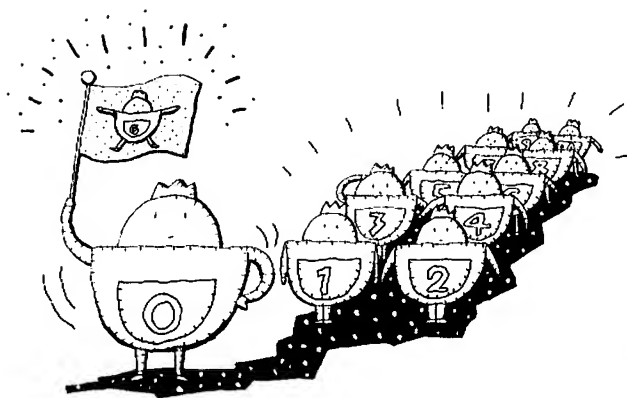
```
nprint(a, 5);
```

と書けばよいのです。さきほど述べたように、配列名 `a` はその先頭要素へのポインタですから、これで配列の先頭から5個ぶんの連続したデータが表示されるというわけです。

なお、a の代わりに &a [0] を使って、

```
nprint(&a [0] , 5);
```

と書いても同じ動作になりますが、&a [0] という書き方が「配列の先頭要素のアドレス」という意味あいを持つものに対して、a の場合は「配列全体を代表する値」のニュアンスが強くなります。これはもう主観の問題なのですが、このように直感的にわかりやすい表記が選べることも、C の良さの1つとってよいでしょう。



文字列の操作

さて、前節でポインタの動作と利用法をつらつらと眺めてきましたが、これらの機能をたいへんにうまく使っているのが、実はCの文字列です。Cの文字列は char 型データの配列で表されるため、ポインタを利用すると非常に巧みな操作が可能です。ポインタの扱い方を覚えるには、文字列操作を例にとるのが一番かもしれません。

■ 文字列の正体を調べる

ここまでは、単に文字を並べてダブルクォートで囲んだものが文字列であるという説明をしてきましたが、文字列に対していろいろな操作を行う場合は、そのような外見の問題だけではなく、文字列がどのように構成されているかも知る必要があります。

そこで、文字列操作の方法を紹介する前に、まず文字列の構造について説明しておくことにしましょう。

● 文字列の構造

Cの文字列は、基本的には char 型データの配列の形をとっています。ただし文字列の最後にはかならずヌル文字(文字コード 0 に相当する文字: '¥0')が付け足され、実際の文字数より 1 バイトだけ大きなサイズのメモリがとられます。

たとえば図 10.12 に示すように、“MSX” という文字列をプログラムで使用すると、メモリの中では 4 バイトの領域が確保され、その先頭から順に、M、S、X、の文字コードが格納されます。そして最後には自動的にヌル文字が追加されます。

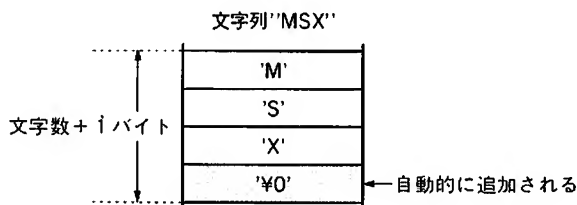


図 10.12 文字列の内部構造

● 文字列はそれ自身へのポインタである

文字列はこのように `char` 型の配列の形をとっているため、前節で述べた配列とポインタの関係を利用すると、さまざまな操作を加えることが可能になります。

ただ、ここで1つ問題があります。普通の配列、たとえば `a[3]` という配列ならば、`&a[0]` あるいは `a` と書くことによって、そのメモリ上のアドレスを知ることができました。ところが文字列には特定の配列名というものがな

いため、同様の方法を使ってアドレスを求められないのです。

そこで、C 言語を開発した人々は、たいへんにうまいことを考え出しました。文字列自身をその文字列へのポインタとして扱おうというのです。

たとえば、`"MSX"` という文字列は、我々にとっては文字列以外のなにものにも見えませんが、コンパイラはこれを `"MSX"` という文字列の先頭へのポインタと解釈するのです。

したがって、ポインタ型の変数 `p` を使って、

```
p = "MSX";
```

と書けば、`"MSX"` という文字列の先頭アドレスが `p` に代入されることになりますし、いままで `puts()` で文字列を表示するのに、ごく当然のように、

```
puts("MSX");
```

などを書いてきたのも、コンパイラにとっては「`"MSX"` という文字列の先頭アドレスを `puts()` に渡す」という意味だったわけです。

■ 文字列操作の実例

ここまでの話をまとめると、文字列には次の3つの重要な性質があることがわかります。

- ① 文字列は char 型データの配列である
- ② 配列の最後にはかならずヌル文字'¥0'が付いている
- ③ 文字列はそれ自身へのポインタと解釈される

文字列を扱う場合は、これらの性質をうまく利用していくことになります。以下に実際の関数定義の例を2つあげてみますので、これを参考に、文字列とポインタを組み合わせたプログラミングの感覚をつかんでください。

● 文字列を表示する関数を作る

まず、関数のパラメータに文字列を使う例として、渡された文字列をそのまま表示する、つまり puts() と同じことを行う関数を作りましょう。図 10.13 がそのプログラムです。

```
newputs(p)
char *p;
{
    while (*p != '¥0') { ..... *p が文字列の終わりでない間
        putchar(*p); ..... *p を表示して
        ++p; ..... p を1進める
    }
}
```

図 10.13 文字列を表示する関数

この関数は、たとえば、

```
newputs("MSX");
```

という形で呼び出します。さきほど述べたとおり、このとき newputs() のパ

ラメータ p には、“MSX”という文字列の先頭へのポインタが渡されます。あとは最後の文字まで、つまりヌル文字‘\0’が出て来る直前まで、p を進めながら 1 文字ずつ表示していけばよいわけです。

●文字列コピーの関数を作る

さて、次は文字列を別の場所にコピーする関数の定義例です。図 10.14 にプログラムを示します。

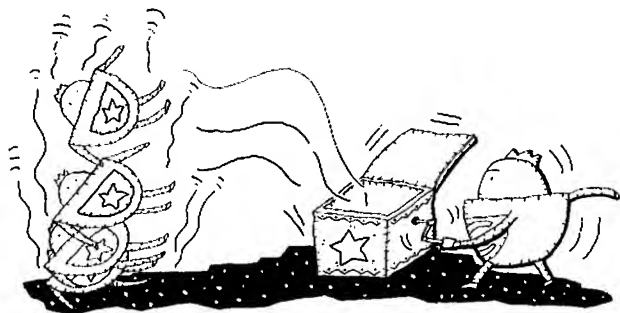
```
copy(p, q) .....文字列qをpの位置にコピーする
char *p, *q;
{
    while (*q != '\0') { .....文字列の最後までループする
        *p = *q; .....1文字コピー
        ++p; .....pを1つ先のアドレスに進める
        ++q; .....qを1つ先のアドレスに進める
    }
    *p = '\0'; .....文字列の最後の印を付ける
}
```

図 10.14 文字列をコピーする関数

この関数を実行するには、コピーを入れる入れ物と、元データになる文字列を指定する必要があります。たとえば、char 型の配列 b[10] に、“MSX”という文字列をコピーするならば、

```
copy(b, "MSX");
```

と書きます。



構造化データを使う

前の節では、主に文字列を使った単純なデータの入出力について述べました。ここでは、もう少しまとまったデータを扱う方法を紹介しましょう。データベースなどを扱う際に便利な構造化データの使い方です。

■ 構造体の使い方

BASIC では、複数のデータを関連付けて扱う手段は配列だけでした。C ではこれに加えて構造体を利用できます。配列は基本的に同じ型のデータをいくつも並べたものですが、構造体は異なる型のデータをまとめて扱うことが可能です。

● 構造体の定義

たとえば名簿を作ることを考えましょう。この名簿には各人の名前と性別と年齢を記載するものとして、次のような3項目を用意することにします。

```
char name [32];      ..... 名前文字列
char sex;            ..... 'M'なら男性, 'F'なら女性とする
int age;             ..... 年齢
```

これらの項目は、3つまとめて1人の人間を表すデータになるわけです。C ではこのようなデータの集まりに、「人間型」とでもいうべき名前を付けて、新しいデータの型を作ることができます。

図 10.15 に、実際にその新しいデータの型を作る方法を示します。

この宣言によって、char 型配列 name [32]、char 型変数 sex、int 型変数 age の3者からなる複合データ型が定義されました。

```

struct person {
    char name[32];
    char sex;
    int age;
};

```

図 10.15 struct person 型の宣言

このようにいくつかのデータを組み合わせて作り出したデータを、構造体と呼びます。また、構造体を構成する各要素(ここでは、name [32], sex, age)を、構造体のメンバーと呼びます。

構造体は「struct」の後ろに特有の呼び名を付けて区別します。上の例の構造体ならば、struct person 型ということになります。

実際にこの構造体を使うには、まず struct person 型の変数を用意しなくてはなりません。たとえば、

```
struct person taro;
```

という変数宣言を行うと、struct person 型の taro という変数が利用できるようになります。また、

```
struct person family [3];
```

という宣言を行えば、これは struct person 型の要素 3 個からなる family [] という配列になります。

●構造体の参照と値の代入

構造体を扱う場合は、それぞれのメンバーごとに個別に操作を行う必要があります。ある構造体変数の中の特定のメンバーを指定するには、その構造体変数名の後ろに、ピリオドとメンバー名を続けます。たとえば、

```
taro.age
```

と書くと、「taro の age」が指定されることになります。

さきほど struct person 型を定義するときに、メンバー age は int 型と宣言

しましたから、この `taro.age` も `int` 型の変数と同様に扱えます。taro の年齢を 7 才としたければ、次のような代入文を実行するだけです。

```
taro.age = 7;
```

この「構造体.メンバー名」という形式は、taro のような単独の変数だけではなく、構造体の配列要素に対しても同様に使用できます。たとえば、

```
family [0].sex
```

と書くと、「family [0] の sex」が指定されるわけです。

●ポインタを使った構造体の操作

すでに述べた `int` 型や `char` 型の変数と同様に、構造体もポインタを使って扱うことができます。とくに構造体を扱う関数を作る場合には、構造体へのポインタは不可欠です。なぜなら、MSX-C では構造体のような複合データは、ポインタの形にしなくてはパラメータとして受け渡すことができないからです。

その例として、`struct person` 型の構造体変数の内容を表示する `whatis()` という関数の定義を図 10.16 に示しましたのでご覧ください。

```
whatis(who)
struct person  *who; ..... who は struct person 型へのポインタ
{
    printf("Name: %s\n", (*who).name);
    printf("Sex:  %c\n", (*who).sex);
    printf("Age:  %d\n", (*who).age);
}
```

図 10.16 構造体へのポインタをパラメータとする関数

この関数を使って、たとえば taro のデータを表示するには、

```
whatis(&taro);
```

という関数呼び出しを行えばよいのです。すると `whatis()` のパラメータ `who` には、構造体 `taro` へのポインタが渡されますから、あとは間接演算子「`*`」を使ってそのポインタの指し示す内容を調べ、画面に表示するだけです。

ところで図 10.16 では、`who` の指し示す構造体のメンバーを、ここまでの説明に従って、

```
(* who).name
```

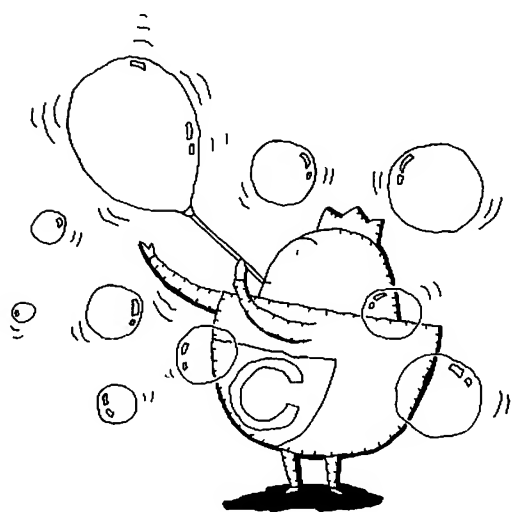
という形式で指定しましたが、このような「ポインタの指し示す構造体のメンバー」を表すには、もう 1 つ次の書式が用意されています。

```
who->name
```

この「`(* who).name`」と「`who->name`」は、まったく同じ意味ですが、実際には後者のほうがよく用いられます。

11章

データの保存と利用



最近、ペーパーレス(Paper-less)環境、という言葉をよく聞くようになりました。要するに、すべての文書がコンピュータのファイルに収められ、それまで会社の中を埋め尽くしていた書類の山がきれいさっぱりなくなってしまうという、理想のオフィスを表す言葉であります。

しかし、机の上から紙の山が消えることのどこが「理想」なのか？ 東南アジアや南米アマゾン流域の緑の資源を食い潰さなくなり、宇宙船地球号の未来が少しでも明るくなるから？ いやいや、そんなきれいごとでは企業の論理に対抗はできません。ペーパーレス環境には、もっと現実的なメリットがあるのです。

データをコンピュータファイルに収めるということは、それをコンピュータが直接処理可能な形態にすることです。紙の上に印刷されたデータは、単にそれを眺めることしかできませんが、コンピュータファイル化されたデータは、それをもとにグラフも描ければ、将来の予測をすることもできます。つまり、データを何倍にも活かして使うことが可能となるのです。

C言語は、このようなファイルの使い回しが非常に得意なプログラミング言語です。そのささやかな第一歩として、この章では文字列データを使ったファイル操作の基本について紹介することにしましょう。

11

1

ファイルを扱うプログラム

ここではファイルへのデータ書き込みと、ファイルからのデータ読み出しの方法について説明します。この2つの処理は、基本的に次のような流れで行われます。

- ① ファイルのオープン
- ② データの読み書き
- ③ ファイルのクローズ

これは BASIC でファイルのデータを操作する場合とまったく同じです。

■ データ読み書きの基本型

ここではファイル入出力の基本的な方法を説明します。

● データ書き込みの基本型

ファイルへの書き込みを行うには、まず目的のファイルを作成し、書き込みの準備を行わなくてはなりません。この操作を「ファイルのオープン」といいます。

ファイルのオープンには、`fopen()`という関数を使用します。たとえば“test”という名前のファイルをオープンするには、次のような形式で `fopen()` を呼び出します。

```
fp = fopen("test", "w");
```

このとき変数 `fp` に代入される `fopen()` の戻り値は、個々のファイルを区別する認識番号のようなもので、正式には「ファイル構造体へのポインタ」と呼ばれる特殊な型のデータですが、本書では単にファイルポインタと呼ぶこ

とにします。ファイルポインタは、次のような形式で宣言しておく必要があります。

```
FILE *fp;
```

以後のファイル操作は、ファイル名ではなく、このファイルポインタを使って行うことになります。たとえばファイルに文字列を書き込むには、

```
fputs(文字列, fp);
```

という関数呼び出しを行います。すでにおなじみの puts() は画面に文字列を表示するものでしたが、この fputs() という関数は、同様な処理を fp で指定されたファイルに対して行うものと考えればよいでしょう。

必要な回数だけ fputs() を実行してデータを書き込み終えたら、ファイルをクローズしなければなりません。ファイルのクローズは、fclose() という関数を使って、次のように行います。

```
fclose(fp);
```

以上の作業を実際のパログラムにまとめたものを、図 11.1 に示します。

```
#include <stdio.h>
main()
{
    FILE *fp; .....ファイルポインタの宣言

    fp = fopen("test", "w"); .....ファイルのオープン
    fputs("data-1\n", fp); .....データの書き込み
    fputs("data-2\n", fp); .....データの書き込み
    fclose(fp); .....ファイルのクローズ
}
```

図 11.1 ファイルへの文字列書き込みプログラム "WTEST.C"

図 11.1 のプログラムを実行すると、"TEST" というファイルが作成されますから、その内容を TYPE コマンドで表示して確かめてください。次のように 2 行ぶんのデータが書き込まれているはずです。


```
A>type test
data-1
data-2
```

図 11.2 図 11.1 の実行結果を確認する

● データ読み出しの基本型

ファイルからのデータ読み出しも、上のデータ書き込みの例と同様にファイルポインタを利用して行います。

まず、データ読み出しのためにファイルをオープンするには、`fopen()`を次のような形で実行します。

```
fp = fopen("test", "r");
```

さきほどは、`fopen()`の第2パラメータには、書き込み(write)モードを示す "w" を指定しましたが、今度は読み出し(read)モードを示すために、"r" というパラメータを指定するわけです。

こうしてオープンされたファイルから文字列の読み込みを行うには、`fgets()`を使用します。これはキーボードからの文字列入力関数 `gets()` のファイル版と考えてください。

ファイルポインタ `fp` で示されるファイルから文字列を読み込み、`char` 型の配列に代入するには、

```
fgets(配列名, 文字数, fp);
```

という関数呼び出しを行います。このとき配列には改行コードまでの1行ぶんのデータが読み込まれます。読み込んだデータの最後には自動的にヌル文字 '\0' が付け加えられますから、これを通常の文字列として扱うことが可能です。なお、読み出そうとした行が指定した文字数にはいきらない場合は、残りのデータは次の `fgets()` で読み込まれます。

データの読み出しが終わったなら、最後にやはり、

```
fclose(fp);
```

を実行して、ファイルをクローズします。

以上の操作を実際プログラムにまとめたものを図 11.3 に示します。これは図 11.1 で作った“test”ファイルの内容を読み出して画面に表示するものです。

```
#include <stdio.h>
main()
{
    char oneline[81];
    FILE *fp; .....ファイルポインタの宣言

    fp = fopen("test", "r"); .....ファイルのオープン
    fgets(oneline, 81, fp); .....1行目のデータの読み出し
    puts(oneline); .....画面に表示
    fgets(oneline, 81, fp); .....2行目のデータの読み出し
    puts(oneline); .....画面に表示
    fclose(fp); .....ファイルのクローズ
}
```

図 11.3 1 行ずつデータを読み出すプログラム

●データ読み出しのチェックポイント

いま見ていただいた図 11.3 は、2 行のデータを含む“test”というファイルを読み出すためのプログラムでしたが、これを元にして、指定した任意のファイルの内容を表示するプログラムを作ってみます。

このとき、次の 2 つの問題を考える必要があります。

- ① 存在しないファイル名が指定されたらどうするか
- ② ファイルの最後をどう認識するか

1 つ目の問題は、`fopen()` の戻り値を調べることで解決できます。指定したファイルがオープンできなかった場合、`fopen()` は `NULL` という記号で表される特別な値(`NULL` ポインタ)を返しますから、次のようなプログラムを書けばよいのです。

```

fp = fopen(ファイル名, "r");
if (fp == NULL) {
    ファイルがオープンできなかった場合の処理;
}

```

2つ目の問題は、fgets()の戻り値で判断できます。ファイルにもう読み出すデータがなくなった状態でfgets()を実行すると、fgets()はやはりNULLを返します。ですから次のようにfgets()の戻り値がNULLかどうかをチェックしていれば、ちょうどファイルのデータをすべて読み終えたところでループを終了させることができます。

```

while (fgets(..., ..., ...) != NULL) {
    読み込んだデータの処理;
}

```

図 11.4 は、上で述べた方法を使って作成した、任意のファイルの内容を表示するプログラム "newtype.c" です。

```

#include <stdio.h>
main(argc, argv)
int  argc;
char *argv[ ];
{
    char oneline[81];
    FILE *fp;

    fp = fopen(argv[1], "w");
    if (fp == NULL) {
        printf("%n7,46 '%s' が 読-7'ノ り'き7474n", argv[1]);
        return;
    }

    while (fgets(oneline, 81, fp) != NULL)
        puts(oneline);

    fclose(fp);
}

```

図 11.4 指定ファイルの内容を画面に表示するプログラム

このプログラムを使用するには、コマンドラインから、

```
A>newtype ファイル名
```

と指定します。

●データ書き込みのチェックポイント

ファイルへのデータ書き込みを行うときも、上で述べたことと同様なエラーチェックの問題はおこります。この場合の問題点は次の2つです。

- ① 不正なファイル名("S.O.S" など)が指定されたらどうするか
- ② ディスクにデータが書き込めなかったらどうするか

1つ目は、やはり `fopen()` の戻り値で判断できます。書き込みモードでファイルをオープンしたとき、もし正常にオープンできなければ `fopen()` は `NULL` を返してきますから、読み出しの場合と同様に次のようなプログラムで対処可能です。

```
fp = fopen(..., ...);
if (fp == NULL) {
    ファイルがオープンできなかった場合の処理;
}
```

2つ目のデータが書き込めないという問題は、まずほとんど、ディスクが一杯になった場合です。これは通常は `fputs()` を実行した時点でおこるのですが、ごくまれに最後のデータを書き込んだ後 `fclose()` を実行した時点でディスクが満杯になったことがわかる場合もありますから、その両方でエラーチェックを行わなければなりません。

`fputs()` は、データが正常に書き込めないと、EOF という記号で表される特別な値を返します。また、`fclose()` も同様に、最後のデータが書き込めないときには EOF を返します。そこで、次のようなプログラムを作ることになります。


```

        if (fclose(fp) == EOF) {
            printf("%nファイル '%s' が クロース ディット/¥n", argv[1]);
            return;
        }

        puts("%nOK¥n");
    }

```

図 11.5 書き込み時のエラーチェックの例

読み書きの手段のバリエーション

ここまで述べたとおり、ファイルに対する文字列の読み書きは、`fgets()` と `fputs()` で行うことができました。これ以外にも MSX-C には、1 文字単位でファイル入出力を行う `getc()` と `putc()`、そして数値の入出力を行う `fprintf()` と `fscanf()` などが用意されています。

これらの関数を使う場合も、ファイルのオープン／クローズは `fgets()`、`fputs()` と同様に行います。それぞれのデータの入出力の手順は、以下のとおりです。

●ファイルへの 1 文字出力 —— `putc()`

`putc()` は、ファイルポインタで示されるファイルに 1 文字ぶんの `char` 型データを書き込みます。正常に書き込めなかった場合は、`EOF` を返します。`putc()` の一般的な使い方は次のようになります。

```

if (putc(c, fp) == EOF) {
    データが書き込めなかった場合の処理;
}

```

●ファイルからの 1 文字入力 —— `getc()`

`getc()` は、ファイルポインタで示されるファイルから 1 文字ぶんのデータを読み出し、その文字コードを戻り値として返します。また、ファイルの最

後までデータが読み出されている場合、さらに読み出しを行うと、`getc()`は EOF を返します。`getc()`を使ってファイルの最後まで読み出すループを作ると次のようになります。

```
int c;
while ((c = getc(fp)) != EOF)
    putchar((char)c);
```

ここで、`getc()`の戻り値を受け取る変数が `int` 型で宣言されていることに注意してください。これは、EOF の値が `char` 型の表現できる範囲に入っていないためです。

●ファイルへの数値出力 —— `fprintf()`

`fprintf()`は、画面に対して `printf()`が行うのとまったく同様な動作を、ファイルに対して行います。データが正常に書き込めなかった場合は EOF を返します。

`fprintf()`によって変数 `i` の値をファイル `fp` に出力するには、次のような関数呼び出しを行います。

```
fprintf(fp, "%d", i);
```

要するに、形式としては `printf()`の1番目のパラメータにファイルポインタが追加されただけです。これ以外にも、`printf()`が画面に対して表示できることはすべて、`fprintf()`を使ってファイルに出力することが可能です。

●ファイルからの数値入力 —— `fscanf()`

`fprintf()`が `printf()`の出力を画面からファイルに振り向けたものならば、`fscanf()`は `scanf()`という関数の入力をキーボードからファイルに振り向けたものに相当します。データがない場合には、`fscanf()`は EOF を返します。

この関数でファイル `fp` から変数 `i` に整数データを読み込むには、

```
fscanf(fp, "%d", &i);
```

という呼び出しを行います。

■ 特別なファイルの利用法

前の項では、ディスク上のファイルに対する入出力の方法を紹介しましたが、プリンタ出力や画面出力、キーボード入力などのいわゆるデバイスについても、まったく同様の方法でデータ入出力のプログラムを書くことができます。

● プリンタへの出力 —— “prn” という名の特殊ファイル

これは MSX-C というよりも、MSX-DOS がもともと持っている機能なのですが、“prn” というファイルに書き込みを行うと、ディスク上の “prn” というファイルではなく、プリンタに対してデータが出力されます。このことを利用すると、プリンタへの打ち出しのプログラムが作成できます。

図 11.6 に、プリンタに A から Z までの 26 文字を打ち出すプログラム例を示します。まず、

```
fp = fopen("prn", "w");
```

という呼び出しでプリンタをオープンしたあと、ここでは `putc()` を使っていますが、`fputs()` や `fprintf()` などでもデータを出力してもかまいません。

```
#include <stdio.h>
main()
{
    FILE *fp;
    char c;

    fp = fopen("prn", "w"); .....プリンタをオープンする
    for (c = 'A'; c <= 'Z'; ++c)
        putc(c, fp);
}
```

図 11.6 プリンタのオープン

● コンソール入出力 —— “con” という名の特殊ファイル

プリンタからデータを読み出すことはできませんから、上記の “prn” デバイスファイルは書き込みモードでしかオープンする意味はありません。

一方 “con” という名前で表される特殊ファイルは、読み出し／書き込みの両方向でオープンすることが可能です。これは、読み出しモードの場合はキーボード入力、書き込みモードの場合は画面出力が指定されます。

たとえば、“con” ファイルをオープンして、

```
fp = fopen("con", "w");
fputs("test¥n", fp);
```

という書き込みを行うと、これはちょうど、

```
puts("test¥n");
```

を実行した場合と同様に、画面に対して文字列 “test” が出力されることになります。

ただし、“con” に対する fputs() と通常の puts() は、まったく同じものというわけではありません。それは、puts() のほうは単なる画面ではなく、「標準出力」にデータを出力するものだからです。

● stdin と stdout

C には、プログラムを作る上でたいへんに便利な「標準入力」、「標準出力」と呼ばれる概念が存在します。

この標準入出力は、通常はコンソール入出力、すなわちキーボード入力と画面出力なのですが、プログラムの実行時にリダイレクションを行うと、そのリダイレクション先に指定されたファイルを対象としてデータを操作するようになります。

この2つには、プログラムを起動した時点で次のような名前のファイルポインタが割り当てられます。

stdin	……	標準入力
stdout	……	標準出力

これらに対するファイルのオープンは自動的に行われるため、プログラム中で実行する必要はありません。たとえば、標準出力に“data”というデータを書き出すプログラムは、図 11.7 のようになります。

```
#include <stdio.h>
main()
{
    fputs("data", stdout);
}
```

図 11.7 標準出力へのデータ書き込みプログラム

実行してみるとわかるように、このプログラムは通常は画面にデータを表示するのですが、出力リダイレクションを行うと、リダイレクション先のファイルにデータを書き込みます。

これは、実際のところ、

```
puts("data");
```

の動作とまったく同じものです。つまり、puts()とは「標準出力に対する fputs()」の省略型にほかならないわけです。

これ以外にも MSX-C では、表 11.1 に示した左右の関数は、双方まったく同じ動作を行います。

入出力先を明示的に指定	入出力先を省略
fputs(..., stdout);	puts(...);
putc(..., stdout);	putchar(...);
fprintf(stdout, ..., ...);	printf(..., ...);
fgets(..., ..., stdin);	gets(..., ...);
getc(stdin);	getchar();
fscanf(stdin, ..., ...);	scanf(..., ...);

表 11.1 標準入出力を使った動作の同じ関数の対応表

● stderr の利用法

標準出力を使うと、リダイレクションの状態によって、データの出力先を画面にするかファイルにするか、自動的に振り分けてくれるのは非常に便利なのですが、1つだけ困る場合があります。

それは、エラーメッセージの表示です。エラーメッセージというのは、そもそも目に見えるように表示されてこそ意味があるわけですが、puts()やprintf()を使ったプログラムでは、出力リダイレクションを行ったとき、エラーメッセージまでファイルに書き込まれ画面に現れてくれません。

この問題を避けるには、標準出力とは別に、もう1つ“con”ファイルを書き込みモードでオープンして、画面に表示したいメッセージはつねにそちらへ出力すればよいはずです。

実はそのために、Cにはあらかじめ標準エラー出力というのが用意されています。これは stderr というファイルポインタで表され、stdin や stdout と同様に、いちいちファイルをオープンしなくとも利用することが可能です。

図 11.8 は、この標準エラー出力の使用例です。このプログラムをコンパイルし、適当なファイルに出力をリダイレクトしてみてください。puts() で出力するデータは指定したファイルに書き込まれますが、「データが書き込めません」というエラーメッセージは、出力リダイレクションを行っているにも関わらず、画面に表示されるはずです。

```
#include <stdio.h>
main()
{
    unsigned i;

    for (i = 1; i <= 60000; ++i) {
        if (puts("test-data%i") == EOF) {
            fputs("#nて'た だ' だきまなて%#n", stderr);
            return;
        }
    }
    puts("OK%i");
}
```

図 11.8 標準エラー出力の利用

Appendix 1

MSX-C エラーメッセージ

● MSX-C CF コマンドエラーメッセージ

- **array of function** 関数の配列は許されない
- **bad #???** #コマンドが間違っている(#コマンドが式中にあるか、プリプロセッサでないコマンドを指定した)
- **bad # <プリプロセッサ命令> statement** # <プリプロセッサ命令> が間違っている
- **bad abstract declarator** 抽象宣言子が正しくない
- **bad assignment** 代入が間違っている
- **bad cast** キャスト演算が間違っている(配列、構造体、共用体に対してはキャストできない)
- **bad character** ソースプログラム中に不適当な文字(コントロール文字など)がある
- **bad condition** 条件式が不正である
- **bad conditional expression (missing '?')** 条件式で3項演算子に '?' がない
- **bad constant expression** 定数式が必要なところに変数や関数が使われている
- **bad drive :** ドライブの指定が間違っている
- **bad element** 文が存在しない、あるいは不正である(ラベルや while の後ろなど)
- **bad element encountered externally** 関数の外側に不正な文がある
- **bad indirection** 間接指定が間違っている(ポインタに*を多く付けすぎた)
- **bad number** 数値表現に範囲外の文字が使われている(8進数に0~7以外の文字を使った場合など)
- **bad option: <文字>** オプションの指定が間違っている('ー<文字>'というオプションは存在しない)
- **bad parameter list** 引数リストが間違っている(引数付きマクロの定義中、仮引数数が不正か、仮引数の区切りが','でない)
- **bad parameter type** 関数呼出しの際の引数の型が数値型ではない(配列、構造体、共用体を渡すにはポインタを使う)
- **bad parameter type ' <仮引数> '** 関数定義の際の <仮引数> の型が数値型ではない
- **bad storage class** 指定の記憶クラスはここでは使用できない
- **bad switch expression** switch 文に誤りがある(switch の式には数値型の結果が必要)
- **bad table ratio** 'ーf'オプションで指定された比率が正しくない
- **bad type in cast** キャストする型が間違っている

- **'break' outside switch/break** break が switch 文の外にある
- **cannot initialized** 自動(auto)変数の配列を初期化をしようとしている
- **cannot make** [<file>] <file> が作成できない
- **cannot open** [<file>] ファイル <file> がオープンできない
- **'case' after default** default より後ろに case がある (MSX-C では default より後ろに case を書くことはできない)
- **'case' outside switch** switch の外側に case がある
- **conflict definition** '識別子' '識別子' の宣言が一致しない (2 つ以上の外部・前方参照の宣言で、関数や変数の宣言が一致していない)
- **conflicting number of macro parameter** マクロの引数の数が合わない
- **'continue' outside loop** while、for などのループの外に continue がある
- **'default' appeared twice** 1 つの switch 文で default が 2 度使われている
- **'default' outside switch** default が switch 文の外にある
- **disk full** ディスクがいっぱいである
- **'double' not supported** double 型は扱えない
- **duplicate 'case' label** switch で case の定数値の同じものが 2 つ以上ある
- **duplicate label** 'ラベル' '同一関数内でラベル' が 2 か所以上存在する
- **duplicate member** 同一の構造体、共用体の中に同じメンバ名が存在する
- **duplicate storage class** 記憶クラスが 2 つ以上指定されている
- **duplicate tag** 'タグ' '同じタグ' で構造体や共用体が定義されている
- **duplicate type specifier** 変数の型が 2 つ以上指定されている
- **{ expected** '{' が必要
- **) expected** ')' が必要
- **: expected** ':' が必要
- **: expected** ':' が必要
- **] expected** ']' が必要
- **'float' not supported** float 型は扱えない
- **function cannot appear** 関数は指定できない
- **function expected** 未定義の関数を呼び出している
- **function returns structured data** 関数は構造体を返すことができない
- **hash table over flow** ハッシュテーブルがオーバーフローした
- **heap over flow** ヒープがオーバーフローした
- **improper function mode** ファンクションモードが不適当である (nonrec、recursive が変数に対して使われている)
- **# include too nested** # include のネストは 4 重まで
- **known dimension expected** 配列の大きさは定数で指定しなければならない
- **'long' not supported** long 型は扱えない

- **l-value required** 左辺値が必要(記憶位置を示す式を左辺値と呼ぶ、値を代入するには変数の名前など、記憶位置を示す式が必要である)
- **missing '}' '}'** '}'が必要
- **missing condition** 条件が必要
- **missing identifier** 識別子(変数名、関数名、配列名など)が必要
- **missing member name** 構造体変数や構造体変数へのポインタに続くメンバ名がない
- **missing operator (or semicolon)** 演算子か ';' が必要
- **missing quote** シングルクォートまたはダブルクォートが必要
- **missing tag** タグ名が必要
- **not appear in parameter list** '〈仮引数〉' '〈仮引数〉' は引数リストにない
- **pointer type mismatch - use cast** ポインタの型が合っていないのでキャスト演算子を使え
- **pool over flow** ブールがオーバーフローした(「-d」オプションでワークテーブルの比率を修正せよ)
- **precedence error** 優先順位が正しくない
- **redeclaration of '〈識別子〉'** '〈識別子〉' が再定義されている
- **sizeof(func) not permitted** sizeof(func)は許可されていない
- **sorry, too small memory** メモリ不足によりコンパイル不可能
- **stac over flow** stacが足りない(「-d」オプションでワークテーブル比率を修正せよ)
- **static function '〈関数〉' not defined** スタティックな関数〈関数〉は定義されていない
- **symbol table over flow** シンボルテーブルが足りない(「-d」オプションでシンボルテーブルの比率が大きくなるよう修正せよ)
- **syntax error** この文または前の文が正しくない
- **too many initializers** 初期化要素の数が多すぎる
- **too much parameters** パラメータが多すぎる(引数付きマクロの引数は 11 まで)
- **type mismatch** 型が合っていない
- **undeclared function '〈関数〉'** '〈関数〉' は宣言されていない
- **undeclared identifier '〈識別子〉'** '〈識別子〉' 未定義の〈識別子〉を使用している
- **undeclared member name '〈メンバ〉'** '〈メンバ〉' が宣言されていない
- **undeclared parameter '〈仮引数〉'** '〈仮引数〉' 仮引数が宣言されていない
- **undefined label '〈ラベル〉' in function '〈関数〉'** '〈関数〉' には〈ラベル〉が定義されていない
- **undefined struct/union** 未定義の struct または union 型として宣言されている
- **unexpected eof in this comment** コメントの途中でファイルが終わっている
- **unexpected eof inside function '〈関数〉'** '〈関数〉' が完結する前にソースファイルが終わっている
- **unexpected eof** ディスク上にテンポラリファイルのための十分な領域がない

- **useless expression** 演算結果が使われていない
- **'while' expected** whileがない(do whileのwhileがループのブロックが終わっても見つからない)

● CG コマンドエラーメッセージ

- **bad option:** <文字> 存在しないスイッチ'<文字>'が指定されている
- **cannot make:** <ファイル> <ファイル> が作成できない
- **cannot open:** <ファイル> <ファイル> が見つからない
- **function overflow in "** <関数> **" ...specify less than** <数> **by -r** <関数> でコード生成用の領域がなくなった、'-r'オプションで<数>より小さい数を指定し直せ
- **illegal .tco file at "** <関数> **" <行> : <桁>** .tco ファイルが壊れてる(再度、CFから始める)
- **sorry, too small memory** メモリ不足によりコンパイル不可能
- **stack overflow** スタックが足りない
- **symbol table overflow in "** <関数> **" ...specify more than** <数> **by -r** <関数> でシンボルテーブルがなくなった、'-r'オプションで<数>より大きい数を指定し直せ
- **warning: >> more than 8 bit in "** <関数> **"** 警告: 8ビット以上右シフトしている
- **warning: << more than 8 bit in "** <関数> **"** 警告: 8ビット以上左シフトしている
- **warning: << more than 16 bit in "** <関数> **"** 警告: 16ビット以上左シフトしている
- **warning: too big char constant** <数> **in "** <関数> **"** 警告: キャラクタ定数の値が大きすぎる

Appendix 2

VRAM ディスクの使い方

以下に第2章で述べた VRAM ディスクのプログラムを紹介します。なお、このプログラムは MSX-DOS1 の上でしか利用できません。

● “RAMDSK.COM” の作成方法

プログラムは 16 進ダンプリストの形式で掲載しました(図 A.1)。これを COM ファイルに変換するには、リスト A.2 に示す “MAKECOM.C” をコンパイルして使用します。最終的に VRAM ディスクプログラム “RAMDSK.COM” を手に入れるまでの手順は以下のとおりです。

```
A>med makecom.c .....*MAKECOM.C*を入力する
|
A>cc makecom .....コンパイルして“MAKECOM.COM”を作る
|
A>med ramdisk.x .....*RAMDISK.X*を入力する
|
A>makecom ramdisk.x ramdisk.com .....“RAMDISK.COM”を作る
```

図 A.1 RAMDISK.COM を手に入れる手順

● VRAM ディスクの利用法

MSX-DOS のコマンドラインから、

```
A>ramdisk -f
```

と入力すると、VRAM ディスクが C ドライブに割り当てられ、初期化されます。通常はアドレス 08000h から 1FFFFh までの 96K バイトの VRAM が使

用されますが、次の形式で、VRAM に割り当てる先頭のアドレスと使用するメモリのサイズを指定することもできます。

```
A>ramdisk -f -t90 -s170 ← 09000h 以降の 17000h バイトを使用
                        (100h バイト単位の 16 進数で指定する)
```

● VRAM ディスクの環境設定

この VRAM ディスクは、起動するときに MSX-DOS の動作環境をリセットするため、“AUTOEXEC.BAT”の中で使うと、それ以降のバッチコマンドが無視されてしまいます。リスト A.3 に示す AUTOKEY コマンドは、それを避けるために使用するものです。これもやはりエディタで入力してから、MAKECOM コマンドで次のように COM ファイルに変換してください。

```
A>makecom autokey.x autokey.com
```

この AUTOKEY コマンドは、DOS のコマンドをセミコロンで区切ってパラメータとして与えると、それらを自動的に順次実行します。たとえば、

```
A>autokey mode 40;dir;
```

と入力すると、まず「mode 40」、続いて「dir」が実行されます。

この機能を使い、下図のような“AUTOEXEC.BAT”および“INITDSK.BAT”を作るとよいでしょう。“AUTOEXEC.BAT”が VRAM ディスクの起動を行い、それ以降のファイルコピー作業は“INITDSK.BAT”が行うわけです。

```
autokey ramdisk -f;initdisk; .....VRAMディスクを起動して INITDSKを実行
```

図 A.2 AUTOEXEC.BAT

```
copy a:command.com c: .....これは絶対に必要
copy a:med.com c: .....以下VRAMディスクに置きたいファイルを
c: .....適当にコピーする
```

図 A.3 INITDSK.BAT

```

0100 C3 B7 04 3A FC FA E6 06 0F 47 04 3A 63 07 4F 3A 721
0110 E3 08 81 28 02 38 11 10 05 FE 21 D8 18 0A 10 05 532
0120 FE 81 D8 18 03 B7 C8 D0 11 66 02 C3 81 01 21 80 840
0130 00 46 23 04 CD FD 01 D8 CD 08 02 28 F7 FE 3F 28 79B
0140 3D FE 2F 20 02 18 04 FE 2D 20 33 CD 50 01 18 E4 680
0150 CD FD 01 38 29 CD 17 02 FE 46 28 16 FE 52 28 18 774
0160 FE 54 28 20 FE 53 28 44 11 79 02 0E 09 CD 05 00 62C
0170 18 0C 3E EF 32 61 07 C9 3E FF 32 62 07 C9 11 E9 7CF

0180 02 C3 D6 04 CD FD 01 38 F5 CD 20 02 38 F0 CD D9 9D4
0190 01 7A FE 02 30 0C CB 43 20 0D 1F CB 1B 7B 32 E3 717
01A0 08 C9 11 8A 02 18 DA 11 A2 02 18 D5 CD FD 01 38 7A5
01B0 CD CD 20 02 38 C8 CD D9 01 7A FE 02 30 16 B7 20 8AA
01C0 0A 7B FE 0C 30 05 11 D3 02 18 B6 CB 3A CB 1B 7B 79E
01D0 32 63 07 C9 11 BF 02 18 A8 CD F5 01 16 00 5F CD 7CC
01E0 FD 01 38 21 CD 20 02 D8 CD F5 01 EB 29 29 29 850
01F0 EB B3 5F 18 EA D6 30 FE 0A D8 D6 07 C9 10 02 37 9C4

0200 C9 7E 23 B7 C9 04 2B C9 FE 09 C8 FE 0D C8 FE 0A A8C
0210 C8 FE 20 C8 FE 3B C9 FE 61 D8 FE 7B D0 D6 20 C9 CFF
0220 CD 17 02 FE 30 D8 FE 3A 38 09 FE 41 D8 FE 47 38 A19
0230 02 37 C9 B7 C9 2E 00 67 16 64 CD 45 02 16 0A CD 7C2
0240 45 02 7C 18 15 1E 00 7C BA 38 04 1C 92 18 F9 67 6E6
0250 7B B7 20 06 7D B7 C8 28 04 AF 2C C6 30 5F E5 0E 8F3
0260 02 CD 05 00 E1 C9 42 61 64 20 41 6C 6C 6F 63 61 851
0270 74 69 6F 6E 20 21 0D 0A 24 42 61 64 20 4F 70 74 700

0280 69 6F 6E 20 21 0D 0A 0D 0A 24 54 6F 70 20 61 64 671
0290 64 20 69 73 20 74 6F 6F 20 68 69 67 68 20 21 0D 770
02A0 0A 24 54 6F 70 20 61 64 64 20 62 69 74 20 23 38 724
02B0 20 6D 75 73 74 20 62 65 20 30 20 21 0D 0A 24 53 69F
02C0 69 7A 65 20 69 73 20 74 6F 6F 20 62 69 67 20 21 809
02D0 0D 0A 24 53 69 7A 65 20 69 73 20 74 6F 6F 20 73 7A7
02E0 6D 61 6C 6C 20 21 0D 0A 24 2D 20 53 65 74 20 56 6F1
02F0 52 41 4D 20 64 69 73 6B 20 64 72 69 76 65 72 20 867

0300 2D 0D 0A 20 20 55 73 61 67 65 3A 20 72 61 6D 64 777
0310 73 6B 20 5B 6F 70 74 69 6F 6E 73 5D 0D 0A 20 20 829
0320 5B 6F 70 74 69 6F 6E 73 5D 0D 0A 20 20 20 20 2D 7A8
0330 72 20 20 20 20 20 20 3A 20 52 65 6D 6F 76 65 20 74A
0340 56 52 41 4D 20 64 69 73 6B 0D 0A 20 20 20 20 2D 705
0350 66 20 20 20 20 20 20 3A 20 46 6F 72 6D 61 74 20 759
0360 56 52 41 4D 20 64 69 73 6B 0D 0A 20 20 20 20 2D 725
0370 74 3C 68 65 78 3E 20 3A 20 54 6F 70 20 61 64 64 899

0380 20 28 70 61 72 20 31 30 30 68 29 0D 0A 20 20 20 6C4
0390 20 2D 73 3C 68 65 78 3E 20 3A 20 53 69 74 65 20 844
03A0 28 70 61 72 20 31 30 30 68 29 20 0D 0A 24 56 52 750
03B0 41 4D 20 64 69 73 6B 20 64 72 69 76 65 72 20 61 936
03C0 6C 72 65 61 64 79 20 65 78 69 73 74 2E 2E 2E 0D 925
03D0 0A 24 54 68 65 72 65 20 69 73 20 6E 6F 74 68 69 934
03E0 6E 67 20 74 6F 20 72 65 6D 6F 76 65 2E 2E 2E 0D 8FD
03F0 0A 24 44 72 69 76 65 20 6E 61 6D 65 73 20 61 72 93F

```

0400	65	20	75	73	65	64	20	75	70	2E	2E	2E	0D	0A	24	44	844
0410	65	66	61	75	6C	74	20	64	72	69	76	65	20	73	65	74	A37
0420	74	6C	65	64	20	6F	6E	20	41	3A	20	64	72	69	76	65	99B
0430	2C	0D	0A	61	6E	64	20	56	52	41	4D	20	64	69	73	6B	8C7
0440	20	69	73	20	2E	2E	2E	24	56	52	41	4D	20	44	69	73	880
0450	6B	20	44	72	69	76	65	72	20	56	65	72	20	31	2E	33	946
0460	30	20	66	6F	72	20	4D	53	58	32	0D	0A	20	20	43	6F	84A
0470	70	79	72	69	67	68	74	20	28	63	29	20	31	39	38	37	944
0480	2C	31	39	38	38	20	62	79	20	54	65	4B	0D	0A	0D	0A	7D3
0490	2D	20	73	65	74	74	6C	65	64	20	6F	6E	20	00	3A	20	949
04A0	64	72	69	76	65	20	28	20	24	4B	20	62	79	74	65	73	9D8
04B0	20	29	20	2D	0D	0A	24	CD	2E	01	CD	03	01	3A	63	07	7F2
04C0	CB	3F	3D	32	F2	08	3A	67	F2	FE	C3	3A	62	07	20	0C	B56
04D0	B7	20	11	11	AE	03	0E	09	CD	05	00	C7	B7	28	48	11	962
04E0	D2	03	18	F2	31	00	C1	0E	0D	CD	05	00	F3	21	47	F3	AEC
04F0	35	2A	68	F2	E5	2E	02	11	67	F2	ED	A0	ED	A0	ED	A0	DCF
0500	11	70	F2	ED	A0	ED	A0	ED	A0	11	79	F2	ED	A0	ED	A0	FB0
0510	ED	A0	11	0F	04	0E	09	CD	05	00	F3	E1	2E	00	7E	23	A4D
0520	66	6F	22	4B	F3	18	58	3A	47	F3	FE	07	38	05	11	F2	B7E
0530	03	18	A3	31	00	C1	32	E4	08	3C	32	47	F3	C6	40	32	ADE
0540	9D	04	11	48	04	0E	09	CD	05	00	3A	63	07	CB	3F	CD	9A2
0550	35	02	11	A9	04	0E	09	CD	05	00	3A	C1	FC	21	06	00	94C
0560	CD	0C	00	32	E1	08	3A	C1	FC	21	07	00	CD	0C	00	3C	A88
0570	32	E2	08	CD	68	F3	CD	30	40	22	00	08	CD	6B	F3	EB	D31
0580	21	FA	0A	01	00	06	AF	ED	42	EB	ED	52	6F	3A	62	07	BC6
0590	B7	20	07	24	24	22	4B	F3	25	25	C5	D5	E5	7C	90	4F	C3A
05A0	21	B7	05	5E	23	56	23	7B	B2	28	05	1A	81	12	18	F3	A89
05B0	D1	C1	E1	D5	ED	B0	C9	02	06	08	06	0E	06	11	06	21	BC0
05C0	06	3F	06	45	06	4B	06	5F	06	66	06	A5	06	AB	06	B4	988
05D0	06	C2	06	C5	06	C9	06	D2	06	D5	06	10	08	18	08	1B	A3E
05E0	08	45	08	56	08	5A	08	76	08	93	08	99	08	A6	08	AA	A07
05F0	08	F8	08	00	00	00	00	00	00	00	00	00	00	00	00	00	6F8
0600	3A	62	07	B7	28	09	11	DC	06	CD	C9	F1	C3	A9	06	21	C98
0610	E4	08	7E	87	06	00	4F	EB	21	55	F3	09	73	23	72	11	BCC
0620	02	08	21	67	F2	ED	A0	ED	A0	ED	A0	21	70	F2	ED	A0	F5B
0630	ED	A0	ED	A0	21	79	F2	ED	A0	ED	A0	ED	A0	21	0B	08	FB1
0640	22	68	F2	21	43	08	22	71	F2	21	91	08	22	7A	F2	3E	C33
0650	C3	32	67	F2	32	70	F2	32	79	F2	CD	68	F3	3A	61	07	E99
0660	B7	28	46	DD	21	E4	08	21	00	00	DD	5E	03	DD	56	04	C05
0670	DD	46	0C	19	10	FD	4D	44	21	00	01	11	01	01	36	00	9C1
0680	ED	B0	21	00	01	3E	FE	77	23	3C	77	23	77	11	00	01	B74
0690	DD	E5	CD	58	50	DD	E1	11	00	00	DD	6E	00	DD	66	0C	E30
06A0	CD	20	47	11	1C	07	CD	C9	F1	11	3C	07	CD	62	44	B7	D0D
06B0	28	0B	11	E8	06	CD	C9	F1	CD	4E	54	18	EC	21	00	00	CFD
06C0	22	5D	07	22	5F	07	23	22	4A	07	11	00	01	CD	58	50	9EB
06D0	11	3C	07	21	00	06	CD	B2	47	C3	00	01	20	72	65	6D	B39
06E0	6F	76	65	64	2E	0D	0A	24	0D	0A	49	6E	73	65	72	74	B83
06F0	20	44	4F	53	20	64	69	73	6B	20	6F	6E	20	64	65	66	C0D

```

0700 61 75 6C 74 20 64 72 69 76 65 20 61 6E 64 20 74 CD7
0710 79 70 65 20 61 6E 79 20 6B 65 79 24 2D 20 56 52 C48
0720 41 4D 20 64 69 73 6B 20 66 6F 72 6D 61 74 20 63 CA5
0730 6F 6D 70 6C 65 74 65 20 2D 0D 0A 24 00 4D 53 58 BA6
0740 44 4F 53 20 20 53 59 53 00 00 00 00 00 00 00 965
0750 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 750
0760 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 820
0770 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 770

```

```

0780 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 780
0790 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 790
07A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7A0
07B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7B0
07C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7C0
07D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7D0
07E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7E0
07F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7F0

```

```

0800 00 00 00 00 00 00 00 00 00 00 00 3A E1 F2 21 E4 B12
0810 08 BE 20 EE E1 DD 21 E4 08 3A F9 08 B7 3A E1 F2 0AE
0820 2A 41 F2 C0 BD 20 0E 25 C8 95 20 09 6F 67 22 3F E0A
0830 F2 2D 22 41 F2 3E FF 32 46 F2 CD FA 45 2B 36 00 FB8
0840 23 18 0C 3A E4 08 DD 96 00 20 BA 32 FF F2 F1 CD FDB
0850 6B F3 E5 D5 3A E2 08 4F 3A E3 08 83 5F C5 7B 07 029
0860 07 07 E6 07 ED 79 3E 8E ED 79 AF 47 ED 79 7B 17 FE1
0870 E6 3E ED 79 3A E1 08 4F E3 E3 ED B2 06 00 ED B2 176

```

```

0880 C1 1C 10 D9 ED 41 3E 8E ED 79 CD 68 F3 D1 E1 B7 237
0890 C9 3A E4 08 DD 96 00 C2 08 08 3C 32 FF F2 F1 CD 0E1
08A0 6B F3 E5 D5 3A E2 08 4F 3A E3 08 83 5F 50 7B 07 004
08B0 07 07 E6 07 ED 79 3E 8E ED 79 AF 47 ED 79 7B 17 031
08C0 E6 3E F6 40 ED 79 0D E3 E3 ED B3 06 00 ED B3 0C 1A5
08D0 1C 15 20 DA ED 51 3E 8E ED 79 CD 68 F3 D1 E1 B7 1FC
08E0 C9 98 99 40 02 FE 00 02 0F 04 01 02 00 00 01 30 C63
08F0 04 00 5F 00 01 01 00 FA 08 00 FE FF FF 00 ED B2 EF2

```

リスト A.1 "RAMDSK.X"

```

#include <stdio.h>
main(argc, argv)
int    argc;
char   **argv;
{
    FILE    *rfp, *wfp;
    unsigned ad, sum, data, i, error;

    if (argc != 3) {
        fprintf(stderr, "Usage: pmud hex_file com_file\n");
        return;
    }
    if ((rfp = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "Cannot read '%s'\n", argv[1]);
        return;
    }
    if ((wfp = fopen(argv[2], "wb")) == NULL) {
        fprintf(stderr, "Cannot write '%s'\n", argv[2]);
        return;
    }

    error = 0;
    while (fscanf(rfp, "%x", &ad) != EOF) {
        sum = ad;
        for (i = 0; i <= 15; ++i) {
            fscanf(rfp, "%x", &data);
            putc((char)data, wfp);
            sum += data;
        }
        fscanf(rfp, "%x", &data);
        if ((sum & 0x0fff) != data) {
            printf("Error at line:%04x\n", ad);
            error = 1;
        }
    }
    if (error == 1)
        unlink(argv[2]);
}

```

リスト A.2 "MAKECOM.C"

```

0100 11 80 00 13 1A FE 0D C8 FE 20 28 F7 FE 09 28 F3 7F0
0110 F3 21 F0 FB 22 FA F3 06 28 1A 13 FE 0D 28 0A FE 8B4
0120 3B 20 02 3E 0D 77 23 10 F0 22 F8 F3 FB C9 00 00 733

```

リスト A.3 "AUTOKEY.X"

索引

A

auto 変数 205

B

break 文 138

C

case 定数 : 144

CC.BAT 50

CEND.REL 49

CF 41

CG 43

char 型 92, 160

CK.REL 49

CLIB.REL 49

con 245

continue 文 140

CRUN.REL 49

D

default : 144

do 文 135

E

else 付きの if 文 117

F

fclose() 236

fflush() 109

fgets() 237

fopen() 235

for 文 129

FPC 162

fprintf() 243

fputs() 236

fscanf() 243

G

getche() 105

getch() 103

getc() 242

gets() 110

goto 文 142

I

if 文 116

int 型 92, 157

K

kbhit() 143

L

L80 47

M

main() 75

MSX-CVer1.1 24

MSX-CVer1.2 24

MSX-DOSTOOLS 25
M80 45

N

NULL 238
NULL ポインタ 238

P

Pascal 17
printf() 88
printf()の桁そろえ表示機能 169
printf()の表示指定記号 152
prn 244
putchar() 85
puts() 86

R

return 文 201

S

setbuf() 108
static 変数 205
stderr 247
stdin 245
stdout 245
struct 230
switch 文 143

T

T コード 41

U

unsigned 型 158

V

VOID 型 204
VRAM ディスク 28, 253

W

W.C.C. 39
while 文 134

10 進数 149
16 進数 150
8 進数 150
! 127
!= 121
include <stdio.h> 71, 78
&& 126
++ 133
-- 133
<= 121
== 121
>= 121
¥n 89
¥0 225
|| 126

ア

アドレス演算子 214
入れ子 118
インクリメント演算子 133
インクルード 78
エコーバック 104
エスケープシーケンス 178
エラー 58
エラーメッセージ 59

オブジェクトプログラム 40

力

書き込みモード 237

空ループ 132

関数 189

関数使用宣言 76, 192

関数定義 192

間接演算子 218

偽 125

記憶クラス 205

キャスト演算子 158

空文 132

グラフィック文字 183

グローバル変数 208

構造化言語 17

構造体 229

コメント 80

コントロール文字 175

コンパイラ 14

コンパイル 40

サ

再設定 129

算術演算子 95

実行条件 129

初期設定 129

真 124

ソースプログラム 40

タ

代入演算子 133

代入式 128

多重分岐 143

デクリメント演算子 133

データの型 154

データの型の優先順位 161

ナ

ニューライン文字 89

ヌル文字 225

ハ

パイプ 63

配列変数 93, 111

バース 42

バッファリング 107

パラメータ 62, 198

比較演算子 121

ビット演算子 98

ビットシフト演算子 100

標準エラー出力 247

標準出力 245

標準入力 245

ファイル構造体 235

ファイルのオープン 235

ファイルのクローズ 236

ファイルポインタ 235

ファンクション・パラメータ・チェック

..... 162

複文 119

符号なし整数 155

フラッシュ 109

プリプロセス 42

ブレース記号 119

ブロック 119

ヘッダファイル 33

変数のスコープ 207

索引

ポインタ型のデータ 215

ポインタ渡し 222

マ

メンバー 230

文字定数 85, 150

戻り値 104, 202

ヤ

読み込みモード 237

ラ

ライブラリ関数 72

ライブラリファイル 72

ラベル 142

リダイレクション 63

リロケートブルモジュール 46

レキシカルアナライズ 42

ローカル変数 195, 208

論理演算子 125

論理値 124

■ 参考文献および推薦書籍

本書を執筆するにあたって以下の文献を参考にさせていただきました。

- | | |
|----------------------------|-----------------------|
| ・ MSX-C Ver.1.1 ユーザーズマニュアル | アスキー |
| ・ MSX-C Ver.1.2 ユーザーズマニュアル | アスキー |
| ・ C ハンドブック | 石田晴久 訳 共立出版 |
| ・ MS-C ハンドブック | アスキー書籍編集部編著 アスキー出版局 |
| ・ プログラミング言語 C | B.W.カーニハン D.M.リッチー 共著 |
| | 石田晴久 訳 共立出版 |

本書を読み終えた方々のための次なるステップとして、以下の書籍をおすすめします。

- | | |
|----------------|--------------------|
| ・ C プログラムブック I | 打越浩幸 濱野尚人 梅原系 共著 |
| ・ 入門 C 言語 | 三田典玄 著 |
| ・ 実習 C 言語 | 三田典玄 著 |
| ・ 改訂新版 C 言語入門 | L.ハンコック M.クリーガー 共著 |
| | アスキー出版局監訳 |

以上 4 点いずれもアスキー出版局

プログラム協力 蔭山哲也

MSX-C 入門 上巻

1989年7月11日 初版発行

定価1,450円(本体1,408円)

著 者 きんがらうじ 桜田幸嗣

発行者 塚本慶一郎

発行所 株式会社 アスキー

〒107-24 東京都港区南青山6-11-1スリーエフ南青山ビル

振 替 東京 4 -161144

TEL (03)486-7111 (大代表)

情報 TEL (03)498-0299 (ダイヤルイン)

出版営業部 TEL (03)486-1977 (ダイヤルイン)

本書は著作権法上の保護を受けています。本書の一部あるいは全部について（ソフトウェア及びプログラムを含む）、株式会社アスキーから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。

制 作 株式会社GARO

印 刷 凸版印刷株式会社

編 集 佐藤英一・竹内充彦

本文デザイン 川戸明子

ISBN4-7561-0006-6 C3055 P1450E